第四章 直列リンク機構

作成：Weiwei Wan
修正：Weiwei Wan, Taiki Moriyama

2024 年 春～夏学期

本章は直列リンク機構の座標変換，運動学，逆運動学，および WRS システムの実装を述べます.

1 回転関節

直列リンク機構とは相互に順番に連結された関節によって構成する機構を指します。この節では，直列リンクの基本構成要素である回転関節と，回転関節の直列リンクにおける回転変換について説明します。この節で紹介する変換は，直動関節にも適用することができます。

回転関節は，固定部分（ステーター）と回転部分（ローター）から構成されます。図 1(a) には，黒軸 $\hat{\boldsymbol{n}}$ を中心に回転する回転関節が示されています。濃い灰色の部分はがステーターであり，薄い灰色の部分がローターです。モーターに電流が流れると，ローターはステーターに対して回転します。この構造に基づいて，回転関節は 3 つの座標系によって定義されます。それらはステーターの座標系，ローターの零点位置の座標系，ローターが回転した後の座標系です．図 1(b) にはこれらの座標系を示しています．ステーターとローターの零点位置の座標系は実線，ローターが回転した後の座標系は点線で表します．それぞれの座標系の位置と回転を $(\bar{\boldsymbol{p}}, \bar{\boldsymbol{R}})$, $(\boldsymbol{p}(0), \boldsymbol{R}(0))$, $(\boldsymbol{p}(\theta), \boldsymbol{R}(\theta))$ と記します．回転関節の回転後も関節の位置は変わらないため，$\boldsymbol{p}(\theta) = \boldsymbol{p}(0)$ です。

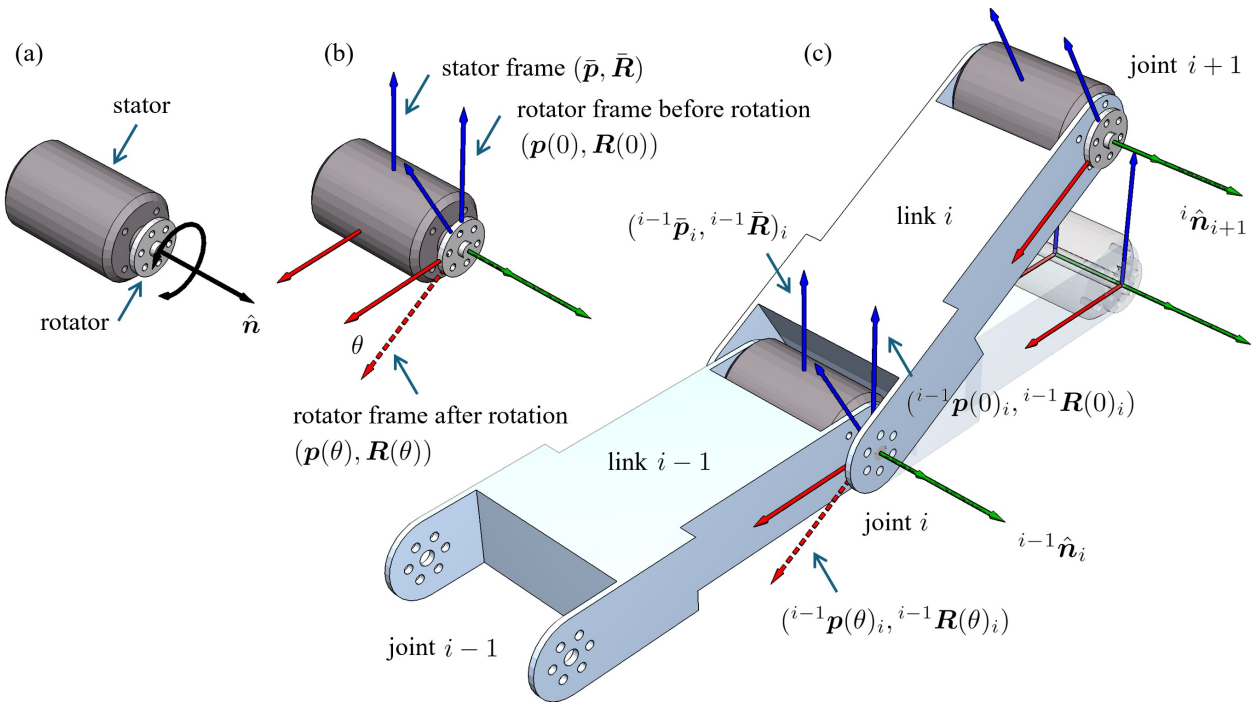

図 1: モーター，関節，リンク座標系の定義と座標関係。 (a) モーターはステーターとローターで構成され，ローターはステーターに対して回転軸を中心に回転します。 (b) モーターのステーター座標系とローター座標系。 (c) 多関節リンクの座標系。

次に，回転関節が直列に接続されている場合を考えます。図 1(c) に示すように，回転関節 i のローター部分はリンク i を介して関節 $i+1$ のステーターに接続されています。接続時に，関節 $i+1$ は関節 i のローターに対してある取り付け姿勢を持っており，この姿勢は関節 $i+1$ のステーター座標系が関節 i のローター座標系におけるローカル姿勢です。これは $({}^{i}\bar{\boldsymbol{p}}_{i+1}, {}^{i}\bar{\boldsymbol{R}}_{i+1})$ として記します．回転軸は ${}^{i}\hat{\boldsymbol{n}}_{i+1}$ と記します．同様に，関節 i のステーター部分は前の関節 $i-1$ のローターに固定され，関節 $i-1$ のローター座標系におけるローカル姿勢 $({}^{i-1}\bar{\boldsymbol{p}}_{i}, {}^{i-1}\bar{\boldsymbol{R}}_{i})$ を持ちます．回転軸は ${}^{i-1}\hat{\boldsymbol{n}}_{i}$ となります．各関節のローターの座標系もステーターと同じように前の関節のローター座標系におけるローカル姿勢を持ちます．関節 i の場合，これらの座標系は $({}^{i-1}\boldsymbol{p}(0)_i, {}^{i-1}\boldsymbol{R}(0)_i)$ と $({}^{i-1}\boldsymbol{p}(\theta)_i, {}^{i-1}\boldsymbol{R}(\theta)_i)$ と記します．関節 i のモーターに電流が流れると，そのモーターのローターが回転し，ローターに接続されたリンク i も回転します。これにより，関節 $i+1$ のステーター，ローター，およびそれに接続された他の関節やリンクも一緒に回転します。

便宜のために，一般的に関節のローターの座標系の零点姿勢とステーターの座標系を同じところに設定します。すなわち，$\bar{\boldsymbol{p}} = \boldsymbol{p}(0), \bar{\boldsymbol{R}} = \boldsymbol{R}(0)$。図 2 は，2 つの座標系が統一された結果を示しています。図では，リンクの具体的な形状は描かれておらず，棒で代用されています。関節 i が回転した後，関節 $i+1$ の変化は前の 1(c) と一致しています。

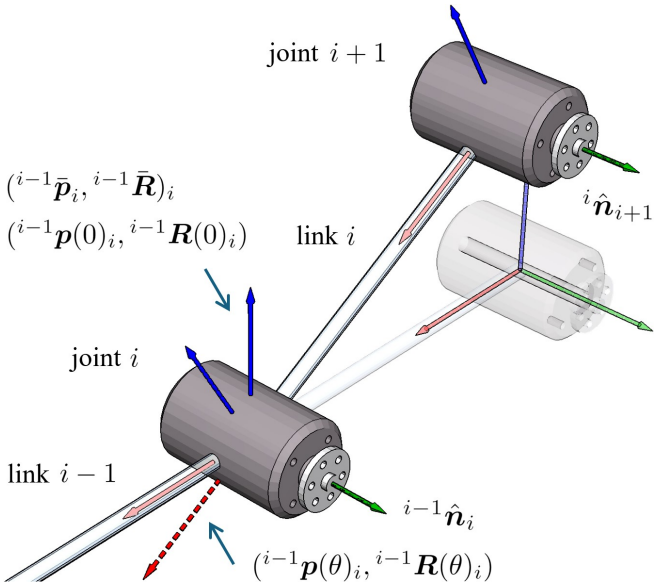

図 2: 表現と解析を簡略化するために，ステーターとローターの座標系を同一位置に定義します。

WRS システムでは，オブジェクト指向プログラミングを使用して，robot_sim/_kinematics/jl.py ファイル内に Joint というクラスを定義しています。リスト 4.1-1 にはこのクラスの初期化関数___init___を示します．この初期化関数では，いくつかのメンバ変数が定義されています。その中で，loc_pos と loc_rotmat は，現在の関節のステーターが前の関節のローター座標系における取り付け姿勢に対応しており，これは上述の ${}^{i-1}\bar{\boldsymbol{p}}_i$ と ${}^{i-1}\bar{\boldsymbol{R}}_i$ に相当します。 loc_motion_ax はモーターのローカル回転軸であり，上述の ${}^{i-1}\hat{\boldsymbol{n}}_i$ に対応しています。現在の関節のローターのローカルな零点姿勢はステーターの座標系と一致していると仮定し，別途メンバ変数を定義していません。メンバ変数_motion_value は現在の関節のローターの回転角度で，上記の θ と対応しています．ローターの回転後のローカル座標はあまり使わないため，別途メンバ変数として保存していません。必要な場合，_motion_value と loc_pos， loc_rotmat を使って計算することができます。局所座標系以外にも，使用の便宜のために，このクラスではローターの原点座標系および回転後の座標系のグローバル座標値を保存するメンバ変数が定義されています。それらはそれぞれ_gl_pos_0, _gl_rotmat_0, _gl_pos_q, _gl_rotmat_q です。最初の 2 つの変数の末尾に「0」が付いているのは，その変数が零点のグローバル座標を保存するために使用されることを意味します。後の 2 つの変数の末尾に「q」が付いているのは，その変数が関節の回転後のグローバル座標を保存するために使用されることを意味します。また，_gl_motion_ax も定義され，モーターのグローバル回転軸を保存します。グローバル座標は，関節の組み立て関係に基づいてプログラムによって自動的に計算されます。具体的な説明は次のセクションで行います。

リスト 4.1-1: Joint クラス

```
    class Joint(object):

        def __init__(self,
                     name="auto",
                     type=rkc.JntType.REVOLUTE,
                     loc_pos=np.zeros(3),
                     loc_rotmat=np.eye(3),
                     loc_motion_ax=np.array([0, 1, 0]),
```

```
                    motion_range=np.array([-np.pi, np.pi])):
            self.name = name
            self.loc_pos = loc_pos
            self.loc_rotmat = loc_rotmat
            self.loc_motion_ax = loc_motion_ax
            self.motion_range = motion_range
            # the following parameters will be updated automatically
            self._motion_value = 0
            self._gl_pos_0 = self.loc_pos
            self._gl_rotmat_0 = self.loc_rotmat
            self._gl_motion_ax = self.loc_motion_ax
            self._gl_pos_q = self._gl_pos_0
            self._gl_rotmat_q = self._gl_rotmat_0
            # the following parameter has a setter function
            self._lnk = Link(name=name)
            # the following parameter should not be changed
            self._type = type
```

WRS システムには，画面上で関節とリンクを描画する機能があります。描画されるのはモーターそのものではなく，ローターとステーターを表す 2 つの円柱です。図 3(a) に示されているように，濃い灰色の円柱がステーターを表し，薄い灰色の円柱がローターを表します。実線の座標軸はステーターの座標系を示し，ローターの零点座標系と一致しています。黒い矢印はモーターの回転軸を示し，黒い矢印の周りの円形の二重矢印は回転範囲を示しています。ローターが一定の角度で回転した後の回転座標系は，図 3(b) に示すように破線の座標軸で表します。図 1(c) と 2 に示されている一部の直列機構とその回転は，WRS の robot_sim/_kinematidcs/jlchain.py の gen_stickmodel でレンダリングすると，図 3(c) のように表現されます。

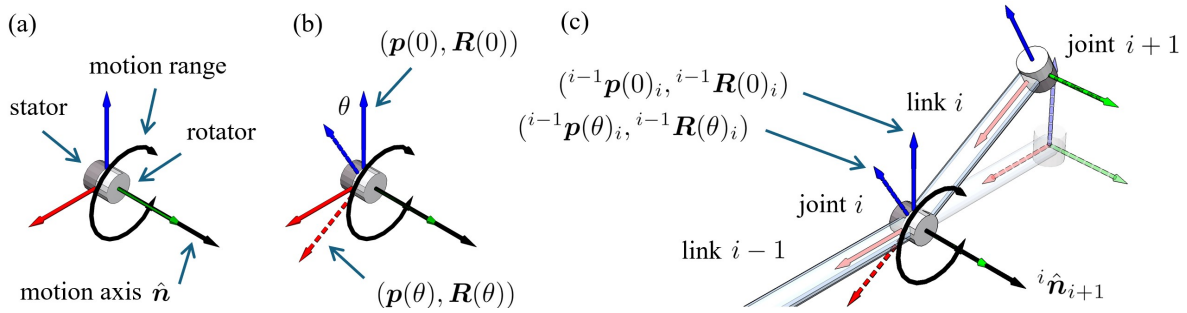

図 3: WRS システムの図示。 (a) モーターとその回転軸。 (b) ローター座標系。 (c) 直列した関節とリンク。

2　全駆動直列リンク機構の運動学と逆運動学

続いて，図 4 のような全駆動直列リンク機構を考えてみましょう．この直列リンクは地面に根を下ろしており，根元の座標系 Σ_O を基準座標系として定義します．直列リンクの先端の位置と回転は $\boldsymbol{p}_e$ と $\boldsymbol{R}_e$ で表されます．$\boldsymbol{p}_e$ は直列リンクの先端の位置を表す 3 次元ベクトルで，$\boldsymbol{R}_e$ は直列リンクの先端の回転を表す 3×3 の回転行列です．WRS は，このような全駆動直列リンク機構を JLChain と呼び，robot_sim/_kinematics/jlchain.py の中で JLChain クラスを定義しています。リスト 4.2-2 にはこのクラスの初期化関数___init___を示します.

リスト 4.2-2: JLChain クラス

```
    class JLChain(object):

        def __init__(self, name="auto", pos=np.zeros(3), rotmat=np.eye(3), n_dof=6):
            self.name = name
            self.n_dof = n_dof
            self.home = np.zeros(self.n_dof)
            # initialize anchor
            self.anchor = rkjl.Anchor(name=f"{name}_anchor", pos=pos, rotmat=rotmat)
            # initialize joints and links
            self.jnts = [rkjl.Joint(name=f"{name}_j{i}") for i in range(self.n_dof)]
            self._jnt_ranges = self._get_jnt_ranges()
```

```
            # default flange joint id, loc_xxx are considered described in it
            self._flange_jnt_id = self.n_dof - 1
            # default flange for cascade connection
            self._loc_flange_pos = np.zeros(3)
            self._loc_flange_rotmat = np.eye(3)
            self._gl_flange_pos = self._loc_flange_pos
            self._gl_flange_rotmat = self._loc_flange_rotmat
            # finalizing tag
            self._is_finalized = False
            # iksolver
            self._ik_solver = None
```

このクラスの内部には，直列リンクの各関節に対応する Joint のリストのメンバ変数 jnts が含まれています。また，根部の固定ノードは Anchor と呼ばれます．Anchor クラスは jl.py の中で定義され，JLChain クラスには Anchor 型のメンバ変数 anchor があります。JLChain の先端の位置と回転 $\boldsymbol{p}_e$ と $\boldsymbol{R}_e$ は，JLChain クラスの gl_flange_pos および gl_flange_rotmat 属性に対応しています。

図 4(a) に示される直列リンクの実装は，リスト 4.2-3 に示されたコードで完成できます。JLChain クラスにはデフォルトの初期局所座標系のパラメータがあり，新しい JLChain を定義する際には，通常これらの局所座標系を変更してニーズを満たす必要があります。コードの 2-34 行目は，局所座標系を再設定しています。再設定が完了したら，JLChain を最終化する必要があります。これにより，可動範囲のクイックインデックス構造が更新され，必要な計算データも準備されます。コードの 35 行目は最終化文です。

リスト 4.2-3: 図 4 に示された直列リンクの実装

```
    jlc = rkjlc.JLChain(n_dof=6)
    # root
    jlc.anchor.pos = np.zeros(3)
    jlc.anchor.rotmat = np.eye(3)
    # joint 0
    jlc.jnts[0].loc_pos = np.array([0, 0, .15])
    jlc.jnts[0].loc_rotmat = np.eye(3)
    jlc.jnts[0].loc_motion_ax = np.array([0, 0, 1])
    jlc.jnts[0].motion_range = np.array([-np.pi * 5 / 6, np.pi * 5 / 6])
    # joint 1
    jlc.jnts[1].loc_pos = np.array([0, 0, .15])
    jlc.jnts[1].loc_rotmat = np.eye(3)
    jlc.jnts[1].loc_motion_ax = np.array([0, 1, 0])
    jlc.jnts[1].motion_range = np.array([-np.pi * 5 / 6, np.pi * 5 / 6])
    # joint 2
    jlc.jnts[2].loc_pos = np.array([0, 0, .15])
    jlc.jnts[2].loc_rotmat = np.eye(3)
    jlc.jnts[2].loc_motion_ax = np.array([0, 1, 0])
    jlc.jnts[2].motion_range = np.array([-np.pi * 5 / 6, np.pi * 5 / 6])
    # joint 3
    jlc.jnts[3].loc_pos = np.array([0, 0, .15])
    jlc.jnts[3].loc_rotmat = np.eye(3)
    jlc.jnts[3].loc_motion_ax = np.array([0, 0, 1])
    jlc.jnts[3].motion_range = np.array([-np.pi * 5 / 6, np.pi * 5 / 6])
    # joint 4
    jlc.jnts[4].loc_pos = np.array([0, 0, .15])
    jlc.jnts[4].loc_rotmat = np.eye(3)
    jlc.jnts[4].loc_motion_ax = np.array([0, 1, 0])
    jlc.jnts[4].motion_range = np.array([-np.pi * 5 / 6, np.pi * 5 / 6])
    # joint 5
    jlc.jnts[5].loc_pos = np.array([0, 0, .15])
    jlc.jnts[5].loc_rotmat = np.eye(3)
    jlc.jnts[5].loc_motion_ax = np.array([0, 0, 1])
    jlc.jnts[5].motion_range = np.array([-np.pi * 5 / 6, np.pi * 5 / 6])
    jlc.finalize()
    # visualize
    base = wd.World(cam_pos=np.array([1.7, 1.7, 1.7]), lookat_pos=np.array
      ([0, 0, .5]))
    jlc.gen_stickmodel(toggle_flange_frame=True, toggle_jnt_frames=True,
      toggle_actuation=True, alpha=.7).attach_to(base)
```

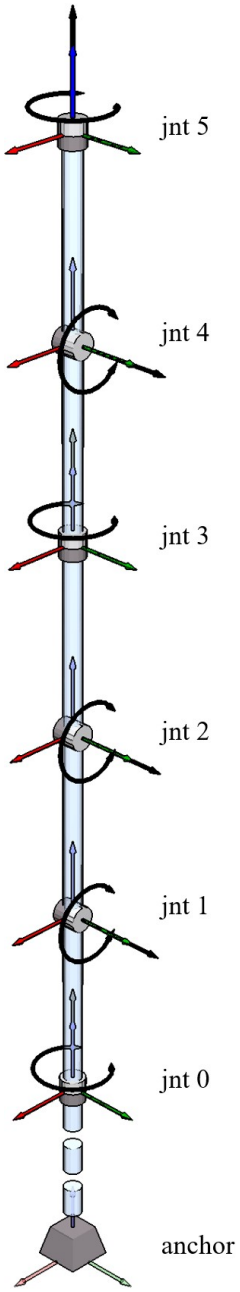

図 4: JLChain

直列リンクに関連する最も重要な 2 つの問題は，順運動学と逆運動学の計算です。与えら

れた関節角度ベクトルに対する直列リンクの先端の位置と回転を求める問題は，順運動学と
呼ばれる．一方，順運動学の逆問題を逆運動学と呼ぶ．逆運動学では，与えられた直列リンクの先端の位置と回転に対する関節角度ベクトルを求めます．ここからは，直列リンク機構の順運動学と逆運動学について説明します．

2.1 順運動学

直列リンク機構の順運動学の本質は，一連の座標変換です。図 4 の直列リンクは回転関節のみを持っていて，各回転関節は，前の回転関節の座標系に取り付けられており，ローカル座標を持っています。根部の基準座標系 Σ_O の位置と回転をそれと $\boldsymbol{p}_O$ と $\boldsymbol{R}_O$ として定義します。関節 0 は基準座標系に取り付けられていますので，モーターのステーターはローカル座標系 $({}^O\bar{\boldsymbol{p}}_0, {}^O\bar{\boldsymbol{R}}_0)$ を持っています．モーターのローターは零点姿勢 $({}^O\boldsymbol{p}(0)_0, {}^O\boldsymbol{R}(0)_0$ と回転後の姿勢 $({}^O\boldsymbol{p}(q_0)_0, {}^O\boldsymbol{R}(q_0)_0$ を持ちます．ここでは，関節の回転を θ ではなく q で表します。 q の下付き文字は，その回転角がどの関節に対応しているかを示します。前節で設定したルールに従い，$({}^O\bar{\boldsymbol{p}}_0, {}^O\bar{\boldsymbol{R}}_0)$ と $({}^O\boldsymbol{p}(0)_0, {}^O\boldsymbol{R}(0)_0)$ を同じく想定します．関節 0 のグローバル座標系は以下の算式で計算できます．

$$\begin{cases} \boldsymbol{p}_0(q_0) = \boldsymbol{R}_O {}^O\boldsymbol{p}(q_0)_0 + \boldsymbol{p}_O \\ \boldsymbol{R}_0(q_0) = \boldsymbol{R}_O {}^O\boldsymbol{R}(q_0)_0 \end{cases} \tag{1}$$

ここで，$\boldsymbol{p}_0(q_0)$ と $\boldsymbol{R}_0(q_0)$ の左上に文字が付いていないから，グローバル座標系と指します．また，ここで，q_0 角度回転後のグローバル座標系の計算方法のみを示しています。零点のグローバル座標系の計算は，q_0 を 0 にするだけで済みます。ステーターのグローバル座標系は零点のグローバル座標系と同じですから，計算方法も同じです。

次に，関節 1 のグローバル座標系の計算を考えます。関節 1 は関節 0 のローターに取り付けられています。上式によって既に関節 0 のローターのグローバル座標系が計算できます。引き続き，上式に従って，関節 1 のローカル座標系を関節 0 のグローバル座標系に重ね合わせると，関節 1 のグローバル座標系を求めることができます。

$$\begin{cases} \boldsymbol{p}_1(q_1) = \boldsymbol{R}_0(q_0) {}^0\boldsymbol{p}(q_1)_1 + \boldsymbol{p}(q_0)_0 \\ \boldsymbol{R}_1(q_1) = \boldsymbol{R}_0(q_0) {}^0\boldsymbol{R}(q_1)_1 \end{cases} \tag{2}$$

関節 1 以降の各関節の座標系も同様に計算できます．一般的に，関節 i の計算は以下の漸化式になります．

$$\begin{cases} \begin{cases} \boldsymbol{p}_i(q_i) = \boldsymbol{R}_{i-1}(q_{i-1}) {}^{i-1}\boldsymbol{p}(q_i)_i + \boldsymbol{p}(q_{i-1})_{i-1} \\ \boldsymbol{R}_i(q_i) = \boldsymbol{R}_{i-1}(q_{i-1}) {}^{i-1}\boldsymbol{R}(q_i)_i \end{cases} & \text{if } i > 0, \\ \begin{cases} \boldsymbol{p}_0(q_0) = \boldsymbol{R}_O {}^O\boldsymbol{p}(q_0)_0 + \boldsymbol{p}_O \\ \boldsymbol{R}_0(q_0) = \boldsymbol{R}_O {}^O\boldsymbol{R}(q_0)_0 \end{cases} & \text{if } i = 0, \end{cases} \tag{3}$$

JLChain の fk 関数は順運動学の計算機能を提供し，そのコアの部分はループを通じて上記の式を実現しています。この関数には update という引数を受け取り，この引数を使用して計算中に各関節およびリンクの_gl_pos と_gl_rotmat メンバー変数を更新するかどうかを制御します。 JLChain を描画する際，システムは各関節およびリンクの_gl_pos と_gl_rotmat メンバー変数を読み取って JLChain の姿勢を描画します。したがって，update 引数は，JLChain の実際の姿勢を更新するかどうかを制御する役割を果たします。もし目標が直接リンクの先端の位置と回転のみを求めることであれば，この引数を False に設定し，各関節およびリンクを更新しないように設定すべきです。逆に，JLChain の姿勢を変更したい場合，update 引数を False に設定し，すべての関節およびリンクの_gl_pos と_gl_rotmat メンバー変数を更新する必要があります．WRS システムでは利便のために，別途 JLChain の goto_given_conf 関数を提供しています。この関数は，update パラメータを True に設定して fk 関数を呼び出すことで，実際に JLChain の姿勢を変更します。図 5 は，図 4 に示され

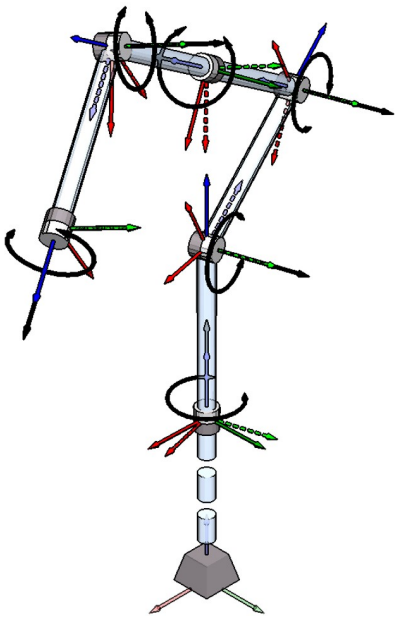

図 5: 順運動学

た JLChain の goto_given_conf() を呼び出した結果を示しています。このとき，各関節の角度が集まった関節角度ベクトル $\boldsymbol{q}$ は次のようになっています。

$$\boldsymbol{q} = [q_0, q_1, q_2, q_3, q_4, q_5] = [\pi/12, -\pi/3, 2\pi/3, \pi/12, \pi/3, 0] \tag{4}$$

ここで，関節角度ベクトル $\boldsymbol{q}$ は各関節角度の値を定義しているため，本質的に JLChain の一つの構型を表しています。そのため，$\boldsymbol{q}$ は「コンフィギュレーション（configuration，構型）」とも呼ばれます。また，$\boldsymbol{q}$ が集まった空間は「関節空間」または「コンフィギュレーション空間」とも呼ばれます。コンフィギュレーション空間の各軸は，それぞれ JLChain の各関節に対応しています。一般的な位置や回転の座標系に対して，これは機構上で定義されるより一般化された座標系であるため,「一般化座標系」と呼ばれます。

2.2 逆運動学

上記の順運動学は関節角ベクトルからエンドエフェクタ座標系への写像問題を解決することです。関数で表すと次のようになります。

$$\begin{bmatrix} \boldsymbol{p}_e \\ \boldsymbol{\xi}_e \end{bmatrix} = f(\boldsymbol{q}) \tag{5}$$

ここで，取扱しやすいため，回転行列 $\boldsymbol{R}_e$ の代わりにあるベクトル形式の表現 $\boldsymbol{\xi}_e$ を使用します．そうすると，関数の左側の $[\boldsymbol{p}_e\ \boldsymbol{\xi}_e]^\top$ はベクトルとなりまして，関数 $f()$ もベクトル演算の関数となります．逆運動学の目標は，この関数 $f()$ の逆関数 $\boldsymbol{q} = f^{-1}([\boldsymbol{p}_e\ \boldsymbol{\xi}_e]^\top)$ を求めることです。一般関数の逆関数の解析解を求めるのは困難です。特別な場合として，関数が線形関数であるときには解析解を求めるのは比較的簡単です。線形関数は一般に $\boldsymbol{y} = \boldsymbol{A}\boldsymbol{x}$ の形式で表すことができます。この場合，$\boldsymbol{x}$ は未知数を表し，$\boldsymbol{A}$ は係数行列を，$\boldsymbol{y}$ は結果のベクトルを表します。逆関数は $\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{y}$ で計算できます．$\boldsymbol{A}$ が可逆かどうかに応じて，擬似逆などの方法を用いて線形関数の逆関数を求めることができます。しかし，直列リンク機構の各関節は順番に接続されているため，順運動学の計算過程では，後ろの関節の角度が前の関節の回転変換に繰り返し乗算され，逐次的な変換が行われます。そのため，順運動学関数は非線形関数となります。従って，逆運動学関数も非線形です。線形関数とは異なり，非線形関数は通常，行列の掛け算で単純に表すことはできません。非線形関数はもっと複雑な構造を持っているため，統一された方法で解くことが非常に難しいのです。非線形関数の一般的な処理方法には，局所線形化やニューラルネットワークによる近似などがあります。本節では，局所線形化を用いて逆運動学を解く方法について紹介します。

2.2.1 局所線形化と逆運動学のニュートン-ラフソン法

局所線形化の基本的な考え方は，非線形関数の定義域のある局所的な微小区間において，定義域と値域の間が局所的に線形であるということです。図 6 はこの基本的な考え方を例にしたもので，2 本の垂直な破線で挟まれた局所的な微小区間内で，定義域と値域に局所的な線形関係が成立していることを仮定しています。逆運動学の問題において，局所の微小区間内では，非線形の順運動学と逆運動学関数は以下のような線形関係で表すことができます。

$$\begin{cases} \begin{bmatrix} \delta\boldsymbol{p}_e \\ \delta\boldsymbol{\xi}_e \end{bmatrix} = \boldsymbol{J}(\boldsymbol{q})\delta\boldsymbol{q} \\ \delta\boldsymbol{q} = \boldsymbol{J}(\boldsymbol{q})^{-1} \begin{bmatrix} \delta\boldsymbol{p}_e \\ \delta\boldsymbol{\xi}_e \end{bmatrix} \end{cases} \tag{6}$$

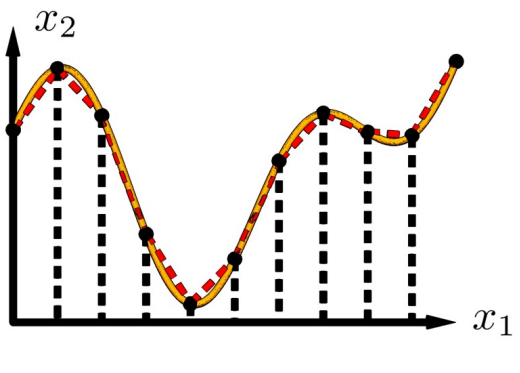

図 6: 局所線形化

ここでの $\delta\boldsymbol{q}$ は，$\boldsymbol{q}$ を始点とする微小区間を指します。$\boldsymbol{J}(\boldsymbol{q})$ は，直列リンクが $\boldsymbol{q}$ にあるときの各関節の微小な移動量と先端姿勢の微小な変化量との線形関係を示す行列です。この行列は，ベクトル値

関数の各変数に対する偏微分を集めたもので，数学的にヤコビ行列
と呼ばれ，$\boldsymbol{J}(\boldsymbol{q})$ と記します。$[\delta \boldsymbol{p}_e \ \delta \boldsymbol{\xi}_e]^\top$ は直列リンクの先端姿勢の微小な変化を表します。

逆運動学を解くために局所線形化を繰り返し利用することができます。その基本的な考え方は，初期値 $\boldsymbol{q}_0$ から始め，直列リンクの先端が目標に到達するまで，繰り返し局所的線形関係を利用して $\boldsymbol{q}_0$ から増加させ，関節値を求めるというものです。このような方法はニュートン-ラフソン法と呼ばれます。以下にその手法を詳述します。

ニュートン-ラフソン法

- 初期化：初期値 $\boldsymbol{q}_0$ を選びます．$\boldsymbol{q}_0$ の選び方はユーザーにお任せするか，あるいは事前に用意したデータから最近傍のものを選択するかなどの方法があります。
- 繰り返し: $\boldsymbol{q}_i, i = 0$ から以下の各サブステップを繰り返し実施します.
  - 誤差の計算：$\boldsymbol{q}_i$ における直接リンク先端の誤差 $[\delta \boldsymbol{p}_e(q_i) \ \delta \boldsymbol{\xi}_e(q_i)]^\top$ を求めます.
  - 終了の判定：誤差が閾値未満であれば終了，そうでなければ次の修正量を計算します.
  - 修正量の求解：$\boldsymbol{q}_i$ におけるヤコビ行列 $\boldsymbol{J}(\boldsymbol{q}_i)$ を計算し，その行列の逆行列を用いて，関節角度の修正量を求めます．算式は以下です.

$$\delta \boldsymbol{q}_i = \boldsymbol{J}(\boldsymbol{q_i})^{-1}[\delta \boldsymbol{p}_e(q_i) \ \delta \boldsymbol{\xi}_e(q_i)]^\top \tag{7}$$

  - 関節角度の修正：修正量を用いて $\boldsymbol{q}_i$ を修正します．$\boldsymbol{q}_{i+1} = \boldsymbol{q}_i + \delta \boldsymbol{q}$

2.2.2 ヤコビ行列の計算

局所線形化とニュートン-ラフソン法の最も重要な問題は，ヤコビ行列の計算とその逆行列の求解です。ヤコビ行列は各関節の微小な変化が直列リンクの先端の姿勢に与える影響を反映しているため，各関節の動きが先端の変動に与える影響を観察・分析することで，この行列の計算方法を導き出します。各関節が微小な回転をする前の関節角度を $\boldsymbol{q}_i$ 回転後の関節角度をと $\boldsymbol{q}_{i+1}$ し，式 (6) に従って末端位置 $\boldsymbol{p}_e$ の変化を以下のように表すことができます。

$$\begin{aligned}
\begin{bmatrix} \delta \boldsymbol{p}_e(\boldsymbol{q}_{i+1} - \boldsymbol{q}_i) \\ \delta \boldsymbol{\xi}_e(\boldsymbol{q}_{i+1} - \boldsymbol{q}_i) \end{bmatrix} &= \boldsymbol{J}(\boldsymbol{q}_i)(\boldsymbol{q}_{i+1} - \boldsymbol{q}_i) = \begin{bmatrix} \delta \boldsymbol{p}_e(\delta \boldsymbol{q}_i) \\ \delta \boldsymbol{\xi}_e(\delta \boldsymbol{q}_i) \end{bmatrix} = \boldsymbol{J}(\boldsymbol{q}_i)\delta \boldsymbol{q}_i \\
&= \boldsymbol{J}(\boldsymbol{q}_i) \begin{bmatrix} \delta q_{i_0} \\ 0 \\ \vdots \\ 0 \end{bmatrix} + \boldsymbol{J}(\boldsymbol{q}_i) \begin{bmatrix} 0 \\ \delta q_{i_1} \\ \vdots \\ 0 \end{bmatrix} + \cdots + \boldsymbol{J}(\boldsymbol{q}_i) \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \delta q_{i_n} \end{bmatrix} \\
&= \begin{pmatrix} \delta \boldsymbol{p}_e(\begin{bmatrix} \delta q_{i_0} \\ 0 \\ \vdots \\ 0 \end{bmatrix}) \\ \delta \boldsymbol{\xi}_e(\begin{bmatrix} \delta q_{i_0} \\ 0 \\ \vdots \\ 0 \end{bmatrix}) \end{pmatrix} + \begin{pmatrix} \delta \boldsymbol{p}_e(\begin{bmatrix} 0 \\ \delta q_{i_1} \\ \vdots \\ 0 \end{bmatrix}) \\ \delta \boldsymbol{\xi}_e(\begin{bmatrix} 0 \\ \delta q_{i_1} \\ \vdots \\ 0 \end{bmatrix}) \end{pmatrix} + \cdots + \begin{pmatrix} \delta \boldsymbol{p}_e(\begin{bmatrix} 0 \\ 0 \\ \vdots \\ \delta q_{i_n} \end{bmatrix}) \\ \delta \boldsymbol{\xi}_e(\begin{bmatrix} 0 \\ 0 \\ \vdots \\ \delta q_{i_n} \end{bmatrix}) \end{pmatrix}
\end{aligned} \tag{8}$$

ここでは，関節数を表すために変数 n を使用します。$\boldsymbol{J}(\boldsymbol{q}_i)$ の最初の 3 行は先端位置と対応していますから，並進ヤコビ子行列と呼ばれます．残りの 3 行は先端の回転に対応しまして，回転ヤコビ子行列と呼ばれます。関節の微小な回転において，各関節の回転が末端に与える変化は回転前のヤコビ行列にのみ依存することを想定しています。つまり，回転中に発生する前の関節の変化が後の関節に与える影響を無視します。そのため，各関節が先端に与える独立した影響をそれぞれ計算し，最後にそれらを統合することで $\boldsymbol{J}(\boldsymbol{q}_i)$ を得ることができます。これから，$\boldsymbol{J}(\boldsymbol{q}_i)$ の詳し算式を導出します.

並進ヤコビ子行列　まず，関節の回転が直列リンクの先端位置 $\boldsymbol{p}_e$ に与える影響を考えます。図 1(a) に示すように，直列リンクの jnt 0 が角度 δq_{i_0} 回転すると，末端位置は次のよう変わります.

$$\boldsymbol{p}_e(\boldsymbol{q}_i + \begin{bmatrix} \delta q_{i_0} \\ 0 \\ \vdots \\ 0 \end{bmatrix}) = \mathrm{rodrigues}(\delta q_{i_0}, \hat{\boldsymbol{n}}_0)(\boldsymbol{p}_e(\boldsymbol{q}_i) - \boldsymbol{p}_0(\boldsymbol{q}_i)) + \boldsymbol{p}_0(\boldsymbol{q}_i) \tag{9}$$

位置の変換量は以下のように表されます.

$$\delta\boldsymbol{p}_e(\begin{bmatrix} \delta q_{i_0} \\ 0 \\ \vdots \\ 0 \end{bmatrix}) = \boldsymbol{p}_e(\boldsymbol{q}_i + \begin{bmatrix} \delta q_{i_0} \\ 0 \\ \vdots \\ 0 \end{bmatrix}) - \boldsymbol{p}_e(\boldsymbol{q}_i) = \mathrm{rodrigues}(\delta q_{i_0}, \hat{\boldsymbol{n}}_0)(\boldsymbol{p}_e(\boldsymbol{q}_i) - \boldsymbol{p}_0(\boldsymbol{q}_i)) + \boldsymbol{p}_0(\boldsymbol{q}_i) - \boldsymbol{p}_e(\boldsymbol{q}_i) \tag{10}$$

δq_{i_0} が小さいとき $\cos(\delta q_{i_0})$ と $\sin(\delta q_{i_0})$ を次のように近似できます.

$$\cos(\delta q_{i_0}) \approx 1, \;\; \sin(\delta q_{i_0}) \approx \delta q_{i_0} \tag{11}$$

この時，式 (10) のロドリゲス公式の項は次のように展開できます.

$$\begin{aligned} &\mathrm{rodrigues}(\delta q_{i_0}, \hat{\boldsymbol{n}}_0)(\boldsymbol{p}_e(\boldsymbol{q}_i) - \boldsymbol{p}_0(\boldsymbol{q}_i)) \\ &= (\boldsymbol{p}_e(\boldsymbol{q}_i) - \boldsymbol{p}_0(\boldsymbol{q}_i))\cos(\delta q_{i_0}) + ((\boldsymbol{p}_e(\boldsymbol{q}_i) - \boldsymbol{p}_0(\boldsymbol{q}_i)) \cdot \hat{\boldsymbol{n}}_0)\hat{\boldsymbol{n}}_0(1 - \cos(\delta q_{i_0})) + (\hat{\boldsymbol{n}}_0 \times (\boldsymbol{p}_e(\boldsymbol{q}_i) - \boldsymbol{p}_0(\boldsymbol{q}_i)))\sin(\delta q_{i_0}) \\ &= \boldsymbol{p}_e(\boldsymbol{q}_i) - \boldsymbol{p}_0(\boldsymbol{q}_i) + (\hat{\boldsymbol{n}}_0 \times (\boldsymbol{p}_e(\boldsymbol{q}_i) - \boldsymbol{p}_0(\boldsymbol{q}_i)))\delta q_{i_0} \end{aligned} \tag{12}$$

式 (10) に代入して整理すると，次の式が貰えます.

$$\delta\boldsymbol{p}_e(\begin{bmatrix} \delta q_{i_0} \\ 0 \\ \vdots \\ 0 \end{bmatrix}) = (\hat{\boldsymbol{n}}_0 \times (\boldsymbol{p}_e(\boldsymbol{q}_i) - \boldsymbol{p}_0(\boldsymbol{q}_i)))\delta q_{i_0} \tag{13}$$

つまり，jnt 0 の微小な回転による直列リンク先端の変化量は，回転角度をヤコビ行列に掛けることで計算できます。同様に，jnt 1 の微小な変化による先端の変化量は次の式で表すことができます。

$$\delta\boldsymbol{p}_e(\begin{bmatrix} 0 \\ \delta q_{i_1} \\ \vdots \\ 0 \end{bmatrix}) = (\hat{\boldsymbol{n}}_1 \times (\boldsymbol{p}_e(\boldsymbol{q}_i) - \boldsymbol{p}_1(\boldsymbol{q}_i)))\delta q_{i_1} \tag{14}$$

jnt 2 以降の計算方法も同様にして導くことができます。すべての関節の式を総合的に計算すると，下式となります.

$$\begin{aligned} \delta\boldsymbol{p}_e(\delta\boldsymbol{q}_i) &= \delta\boldsymbol{p}_e(\begin{bmatrix} \delta q_{i_0} \\ 0 \\ \vdots \\ 0 \end{bmatrix}) + \delta\boldsymbol{p}_e(\begin{bmatrix} 0 \\ \delta q_{i_1} \\ \vdots \\ 0 \end{bmatrix}) + \delta\boldsymbol{p}_e(\begin{bmatrix} 0 \\ 0 \\ \vdots \\ \delta q_n \end{bmatrix}) \\ &= (\hat{\boldsymbol{n}}_0 \times (\boldsymbol{p}_e(\boldsymbol{q}_i) - \boldsymbol{p}_0(\boldsymbol{q}_i)))\delta q_{i_0} + (\hat{\boldsymbol{n}}_1 \times (\boldsymbol{p}_e(\boldsymbol{q}_i) - \boldsymbol{p}_1(\boldsymbol{q}_i)))\delta q_{i_1} + \cdots + (\hat{\boldsymbol{n}}_n \times (\boldsymbol{p}_e(\boldsymbol{q}_i) - \boldsymbol{p}_n(\boldsymbol{q}_i)))\delta q_{i_n} \\ &= \begin{bmatrix} \hat{\boldsymbol{n}}_0 \times (\boldsymbol{p}_e(\boldsymbol{q}_i) - \boldsymbol{p}_0(\boldsymbol{q}_i)) & \hat{\boldsymbol{n}}_1 \times (\boldsymbol{p}_e(\boldsymbol{q}_i) - \boldsymbol{p}_1(\boldsymbol{q}_i)) & \cdots & \hat{\boldsymbol{n}}_n \times (\boldsymbol{p}_e(\boldsymbol{q}_i) - \boldsymbol{p}_n(\boldsymbol{q}_i)) \end{bmatrix} \begin{bmatrix} \delta q_{i_0} \\ \delta q_{i_1} \\ \vdots \\ \delta q_{i_n} \end{bmatrix} \end{aligned}$$

$$= \begin{bmatrix} \hat{\boldsymbol{n}}_0 \times (\boldsymbol{p}_e(\boldsymbol{q}_i) - \boldsymbol{p}_0(\boldsymbol{q}_i)) & \hat{\boldsymbol{n}}_1 \times (\boldsymbol{p}_e(\boldsymbol{q}_i) - \boldsymbol{p}_1(\boldsymbol{q}_i)) & \cdots & \hat{\boldsymbol{n}}_n \times (\boldsymbol{p}_e(\boldsymbol{q}_i) - \boldsymbol{p}_n(\boldsymbol{q}_i)) \end{bmatrix} \delta\boldsymbol{q}_i. \tag{15}$$

式 (8) との照合により，$\boldsymbol{J}(\boldsymbol{q}_i)$ 行列の最初の 3 行の計算式を得ることができます.

$$\boldsymbol{J}(\boldsymbol{q}_i)\delta\boldsymbol{q}_i = \begin{bmatrix} \delta\boldsymbol{p}_e(\delta\boldsymbol{q}_i) \\ \vdots \end{bmatrix} = \begin{bmatrix} \hat{\boldsymbol{n}}_0 \times (\boldsymbol{p}_e(\boldsymbol{q}_i) - \boldsymbol{p}_0(\boldsymbol{q}_i)) & \hat{\boldsymbol{n}}_1 \times (\boldsymbol{p}_e(\boldsymbol{q}_i) - \boldsymbol{p}_1(\boldsymbol{q}_i)) & \cdots & \hat{\boldsymbol{n}}_n \times (\boldsymbol{p}_e(\boldsymbol{q}_i) - \boldsymbol{p}_n(\boldsymbol{q}_i)) \\ \vdots & \vdots & \ddots & \vdots \end{bmatrix} \delta\boldsymbol{q}_i \tag{16}$$

注意事項として，以上のすべての導出は各関節が微小な回転をすることを前提としています。その場合，各関節の回転が直列リンクの先端に与える影響は回転前のヤコビ行列のみに依存しており，先端への影響の計算は各関節の回転による変化をベクトルの加算として取り扱うことが可能です。逆に回転量が大きい場合，各関節の影響が順次伝わることになります。このとき，末端の変化は回転前のヤコビ行列のみに依存しなくなり，計算に大きな誤差が生じることがあります。図 7 は 2 つの関節が大きな角度で回転する例を示しており，末端の変化量とベクトル加算の結果に大きな誤差が生じることが分かります。

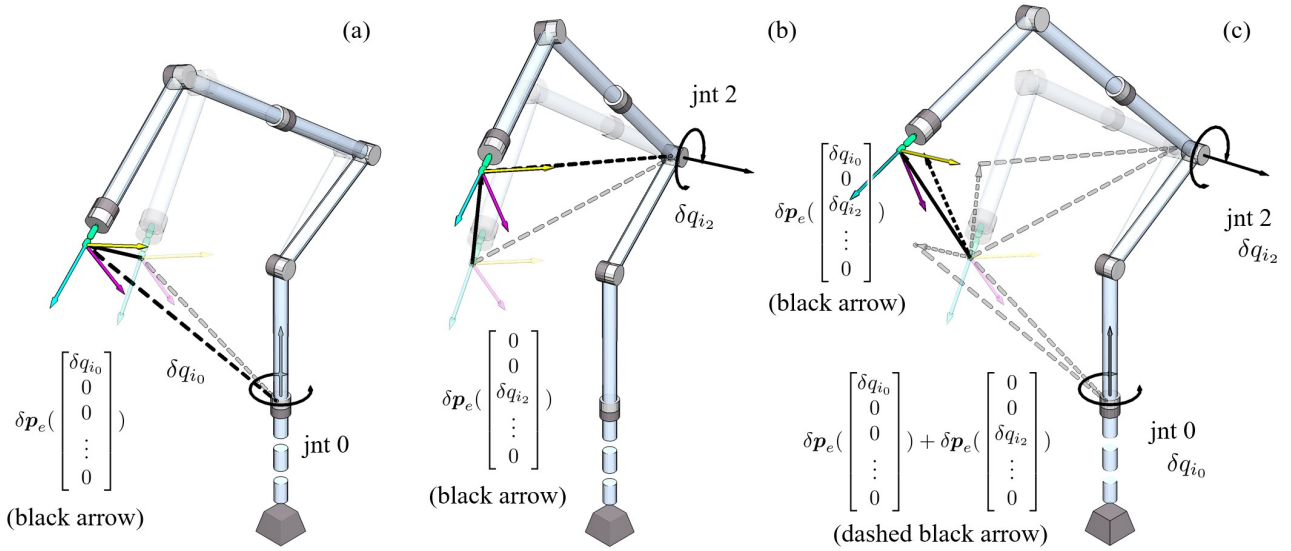

図 7: 式 (16) は各関節が微小な回転をすることを前提としています。関節の回転角が大きい場合,, こちらの図に示すように，末端の位置の移動は累加結果と大きな誤差が生じます。

回転ヤコビ子行列　次に，各関節の回転が直列リンク先端の姿勢に与える影響を分析します。ここで考えるべき問題は，どのようなベクトル形式を用いて回転を表現するかということです。候補としては，オイラー角，回転ベクトル，四元数などがあります。オイラー角の表現は，ジンバルロックや回転間の差の評価は困難などの問題があるため，考慮しません。残りの回転ベクトルと四元数を検討します。

式 (8) の演算により，$\boldsymbol{J}(\boldsymbol{q}_i)$ 行列の 3 行目以降の計算は以下の線形和に満たさないといけないことが分かります.

$$\delta\boldsymbol{\xi}_e(\delta\boldsymbol{q}_i) = \delta\boldsymbol{\xi}_e\left(\begin{bmatrix} \delta q_{i_0} \\ 0 \\ \vdots \\ 0 \end{bmatrix}\right) + \delta\boldsymbol{\xi}_e\left(\begin{bmatrix} 0 \\ \delta q_{i_1} \\ \vdots \\ 0 \end{bmatrix}\right) + \cdots + \delta\boldsymbol{\xi}_e\left(\begin{bmatrix} 0 \\ 0 \\ \vdots \\ \delta q_{i_n} \end{bmatrix}\right) \tag{17}$$

回転ベクトルで回転を表す場合，これは下記の形になります.

$$\delta q_e \hat{\boldsymbol{n}}_e = \delta q_{i_0}\hat{\boldsymbol{n}}_0 + \delta q_{i_1}\hat{\boldsymbol{n}}_1 + \cdots + \delta q_{i_n}\hat{\boldsymbol{n}}_n \tag{18}$$

回転ベクトルの特性からわかるように，回転角度が小さい場合，回転の合成は回転ベクトルの線形和に相当します。したがって，上式が成立します。これにより，回転ベクトルは回転の局所線形化計算に非常に適しています。

皆さんがより理解しやすくするために，次に上記の線形和が回転の合成と等価であることを再度証明します。回転行列を使用する場合，直列リンク先端の姿勢は各関節の回転後，次の式で計算できます。

$$\boldsymbol{R}_e(\delta\boldsymbol{q}_i) = \boldsymbol{R}_e(\begin{bmatrix}\delta q_{i_0}\\0\\\vdots\\0\end{bmatrix})\boldsymbol{R}_e(\begin{bmatrix}0\\\delta q_{i_1}\\\vdots\\0\end{bmatrix})\cdots\boldsymbol{R}_e(\begin{bmatrix}0\\0\\\vdots\\\delta q_{i_n}\end{bmatrix}) \tag{19}$$

回転ベクトルを用いて表す場合，ずロドリゲスの公式を用いて回転行列に変換し，積を計算してからベクトル形式に戻す必要があるためです。すなわち：

$$\boldsymbol{R}_e(\delta\boldsymbol{q}_i) = \text{rodrigues}(\delta q_e,\ \hat{\boldsymbol{n}}_e) = \text{rodrigues}(\delta q_0,\ \hat{\boldsymbol{n}}_0)\cdot\text{rodrigues}(\delta q_1,\ \hat{\boldsymbol{n}}_1)\cdots\text{rodrigues}(\delta q_n,\ \hat{\boldsymbol{n}}_n) \tag{20}$$

ロドリゲス公式の回転角度が小さい場合,

$$\cos(\theta)\approx 1,\ \ \sin(\theta)\approx\theta \tag{21}$$

ですから，ロドリゲス回転公式は次のように近似できます.

$$\text{rodrigues}(\theta,\hat{\boldsymbol{n}}) = \boldsymbol{I} + \sin(\theta)[\hat{\boldsymbol{n}}\times] + (1-\cos(\theta))[\hat{\boldsymbol{n}}\times]^2 \approx \boldsymbol{I} + \theta[\hat{\boldsymbol{n}}\times]. \tag{22}$$

上記の式 (20) に代入すると下の式が出ます.

$$\begin{aligned}\boldsymbol{R}_e(\delta\boldsymbol{q}_i) &= \text{rodrigues}(\delta q_e,\ \hat{\boldsymbol{n}}_e) \approx (\boldsymbol{I} + \delta q_e[\hat{\boldsymbol{n}_e}\times]) = (\boldsymbol{I} + [(\underline{\delta q_e\hat{\boldsymbol{n}_e}})\times])\\ &\approx (\boldsymbol{I} + \delta q_0[\hat{\boldsymbol{n}_0}\times])\cdot(\boldsymbol{I} + \delta q_1[\hat{\boldsymbol{n}_1}\times])\cdots(\boldsymbol{I} + \delta q_n[\hat{\boldsymbol{n}_n})\times])\\ &= \boldsymbol{I} + [(\underline{\delta q_0\hat{\boldsymbol{n}}_0 + \delta q_1\hat{\boldsymbol{n}}_1 + \cdots + \delta q_n\hat{\boldsymbol{n}}_n})\times]\end{aligned} \tag{23}$$

下線部を比較すると，$\delta q_e\hat{\boldsymbol{n}}_e = \delta q_{i_0}\hat{\boldsymbol{n}}_0 + \delta q_{i_1}\hat{\boldsymbol{n}}_1 + \cdots + \delta q_{i_n}\hat{\boldsymbol{n}}_n$ であることがわかり，したがって，上記の回転ベクトルは回転の局所線形化計算に適当である結論が証明されます。

回転ベクトルの代わりに，四元数を使用する場合，式 (25) は次の形になります。

$$\begin{bmatrix}\cos(\delta q_e/2)\\\hat{n}_{e_x}\sin(\delta q_e/2)\\\hat{n}_{e_y}\sin(\delta q_e/2)\\\hat{n}_{e_z}\sin(\delta q_e/2)\end{bmatrix} = \begin{bmatrix}\cos(\delta q_0/2)\\\hat{n}_{0_x}\sin(\delta q_0/2)\\\hat{n}_{0_y}\sin(\delta q_0/2)\\\hat{n}_{0_z}\sin(\delta q_0/2)\end{bmatrix} + \begin{bmatrix}\cos(\delta q_1/2)\\\hat{n}_{1_x}\sin(\delta q_1/2)\\\hat{n}_{1_y}\sin(\delta q_1/2)\\\hat{n}_{1_z}\sin(\delta q_1/2)\end{bmatrix} + \cdots + \begin{bmatrix}\cos(\delta q_n/2)\\\hat{n}_{n_x}\sin(\delta q_n/2)\\\hat{n}_{n_y}\sin(\delta q_n/2)\\\hat{n}_{n_z}\sin(\delta q_n/2)\end{bmatrix} \tag{24}$$

第一行の cos 計算によって，回転角度が非常に小さい場合でも，この式が成り立たないことが容易に確認できます。そのため，四元数は回転の局所線形化計算に適しません.

以上の議論から，順逆運動学の局所線形化計算の過程では，回転を表すために回転ベクトル表現を採用すべきことが分ります。$\boldsymbol{J}(\boldsymbol{q}_i)\delta\boldsymbol{q}_i$ の回転部の計算式は，以下になります.

$$\begin{aligned}\delta\boldsymbol{\xi}_e(\delta\boldsymbol{q}_i) &= \delta\boldsymbol{\xi}_e(\begin{bmatrix}\delta q_{i_0}\\0\\\vdots\\0\end{bmatrix}) + \delta\boldsymbol{\xi}_e(\begin{bmatrix}0\\\delta q_{i_1}\\\vdots\\0\end{bmatrix}) + \cdots + \delta\boldsymbol{\xi}_e(\begin{bmatrix}0\\0\\\vdots\\\delta q_{i_n}\end{bmatrix})\\ &= \delta q_{i_0}\hat{\boldsymbol{n}}_0 + \delta q_{i_1}\hat{\boldsymbol{n}}_1 + \cdots + \delta q_{i_n}\hat{\boldsymbol{n}}_n = \begin{bmatrix}\hat{\boldsymbol{n}}_0 & \hat{\boldsymbol{n}}_1 & \cdots & \hat{\boldsymbol{n}}_n\end{bmatrix}\begin{bmatrix}\delta q_{i_0}\\\delta q_{i_1}\\\vdots\\\delta q_{i_n}\end{bmatrix} = \begin{bmatrix}\hat{\boldsymbol{n}}_0 & \hat{\boldsymbol{n}}_1 & \cdots & \hat{\boldsymbol{n}}_n\end{bmatrix}\delta\boldsymbol{q}_i\end{aligned} \tag{25}$$

これにより，$\boldsymbol{J}(\boldsymbol{q}_i)\delta\boldsymbol{q}_i$ の 3 行目以降は，以下の形になります.

$$\boldsymbol{J}(\boldsymbol{q}_i)\delta\boldsymbol{q}_i = \begin{bmatrix}\vdots\\\delta\boldsymbol{\xi}_e(\delta\boldsymbol{q}_i)\end{bmatrix} = \begin{bmatrix}\vdots & \vdots & \ddots & \vdots\\\hat{\boldsymbol{n}}_0 & \hat{\boldsymbol{n}}_1 & \cdots & \hat{\boldsymbol{n}}_n\end{bmatrix}\delta\boldsymbol{q}_i \tag{26}$$

ヤコビ行列　式 (26) と式 (16) と合わせると，次の $\boldsymbol{J}(\boldsymbol{q}_i)$ の全体的な計算式がもらえます。$\boldsymbol{J}(\boldsymbol{q}_i)$ は $6 \times n$ の行列です。

$$\boldsymbol{J}(\boldsymbol{q}_i) = \begin{bmatrix} \hat{\boldsymbol{n}}_0 \times (\boldsymbol{p}_e(\boldsymbol{q}_i) - \boldsymbol{p}_0(\boldsymbol{q}_i)) & \hat{\boldsymbol{n}}_1 \times (\boldsymbol{p}_e(\boldsymbol{q}_i) - \boldsymbol{p}_1(\boldsymbol{q}_i)) & \cdots & \hat{\boldsymbol{n}}_n \times (\boldsymbol{p}_e(\boldsymbol{q}_i) - \boldsymbol{p}_n(\boldsymbol{q}_i)) \\ \hat{\boldsymbol{n}}_0 & \hat{\boldsymbol{n}}_1 & \cdots & \delta q_{i_n} \end{bmatrix} \tag{27}$$

2.2.3　関節角度の修正量の求解 1：減衰最小二乗法

ヤコビ行列が求められた後，式 (7) を使用して関節角度の修正量を計算し，関節角度を修正することで，直列リンクの先端の姿勢と目標の間の誤差を減少させることができます。これをニュートン-ラフソン法で繰り返し反復することで，最終的な逆運動学の解を得ることができます。式 (7) の計算には逆行列が必要です。逆行列は，ガウス・ジョルダン法（Gauss-Jordan elimination）や LU 分解（LU decomposition）を使って求めることは一般的です。しかし，すべての行列が逆行列を持つわけではありません。ある行列が逆行列を持つためには，次の正則条件を満たす必要があります： (1) 正方行列であること。 (2) 行列の行列式がゼロでないこと。$\boldsymbol{J}(\boldsymbol{q}_i)$ の場合，それらの条件は $\boldsymbol{J}(\boldsymbol{q}_i)$ の列数 $n = 6$ と $\det(\boldsymbol{J}(\boldsymbol{q}_i)) \neq 0$ となります．すなわち，直接リンク機構は 6 つの関節を持ったなければならず，行列式がゼロになるような姿勢にならないようにしなければなりません。一番目の条件は何とか満たせますが，二番目の条件は数値解が反復更新される計算であるため，計算過程でヤコビ行列の行列式がゼロになる状況を避けるのは難しいです。そのため，ヤコビ行列の逆行列が存在しない場合に対応できる修正量の計算法を見つける必要があります。

この問題の本質は線形方程式系の解法の問題です。線形方程式系を解く際には，方程式系の形式に応じて逆行列を解くか，最適な演算で逆行列を近似するかなどの手法があります。ここで，直線の交点問題を例に，線形方程式系の異なる形式とそれらの解法を説明します。

過剰決定系と劣決定系　二次元直線の交点の問題を考えましょう．二次元の直線は以下の方程式で表現できます．

$$a_1 x_1 + a_2 x_2 = b \tag{28}$$

複数の直線の交点は以下の方程式系，または行列とベクトルの形式で表現できます．

$$\begin{cases} a_{11}x_1 + a_{12}x_2 = b_1 \\ a_{21}x_1 + a_{22}x_2 = b_2 \\ \cdots \end{cases} \quad \Rightarrow \quad \begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ \vdots & \vdots \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \end{bmatrix} \Rightarrow \boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \tag{29}$$

行列 $\boldsymbol{A}$ はそのランクに応じて，フルランク，行フルランク，列フルランク，ランク落ちに場合分けされます．それぞれの場合の直線の様子を図 8 に示します．　各場合に対応する行列 $\boldsymbol{A}$ の値は

$$\text{(a)}: \begin{bmatrix} -1 & 2 \\ 0.2 & 1 \end{bmatrix}, \quad \text{(b)}: \begin{bmatrix} -1 & 2 \end{bmatrix}, \quad \text{(c)}: \begin{bmatrix} -1 & 2 \\ 0.2 & 1 \\ 0.4 & 0.2 \end{bmatrix}, \quad \text{(d)}: \begin{bmatrix} -1 & 2 \\ 0.2 & 1 \\ -10 & 20 \end{bmatrix}, \quad \text{(e)}: \begin{bmatrix} -1 & 2 \\ -10 & 20 \end{bmatrix} \tag{30}$$

となります．簡約化すると以下の値になります．

$$\text{(a)}: \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \quad \text{(b)}: \begin{bmatrix} 1 & -2 \end{bmatrix}, \quad \text{(c)}: \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}, \quad \text{(d)}: \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0 & 0 \end{bmatrix}, \quad \text{(e)}: \begin{bmatrix} 1 & -2 \\ 0 & 0 \end{bmatrix} \tag{31}$$

それぞれのランクは行列から簡単に求められます．

$\boldsymbol{A}$ がフルランク（列フルランクと行ランクを同時に持つ）場合，線形方程式系の解は

$$\boldsymbol{x} = \boldsymbol{A}^{-1}\boldsymbol{b} \tag{32}$$

です．$\boldsymbol{A}$ が列フルランクをもつ（同時に行フルランクをもたない）とき，線形方程式系は過剰決定系となります．過剰決定系は方程式の数が未知数の数よりも多いので，$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ を満たす $\boldsymbol{x}$ の解は存在しません．そのため，過剰決

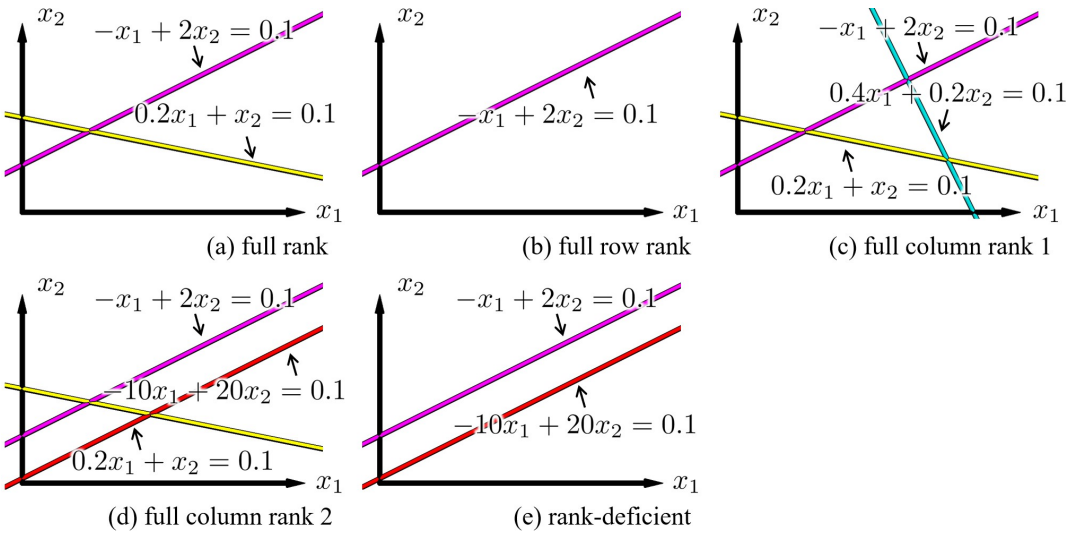

図 8: 行列 $\boldsymbol{A}$ が (a) フルランク，(b) 行フルランク，(c,d) 列フルランク，(e) ランク落ちのときの直線の様子

定系はある誤差を最小にする $\hat{\boldsymbol{x}}$ しか求められません．過剰決定系に対して，$\boldsymbol{A}$ が行フルランクをもつ（同時に列フルランクをもたない）とき，線形方程式系は劣決定系となります．劣決定系は方程式の数が未知数の数より少ないので，$\boldsymbol{Ax} = \boldsymbol{b}$ を満たす $\boldsymbol{x}$ の解は無限個存在します．そのため，劣決定系の一つの解を求める際は，無限個の解の中からある条件を満たす最適な $\hat{\boldsymbol{x}}$ を選ぶ必要があります．

行列 $\boldsymbol{A}$ が異なるランクをもつときの交点の解を，図 8 から直観的に観察してみましょう．図 8(a) には二つの直線があります．このとき，行列 $\boldsymbol{A}$ はフルランクをもっていて，方程式の数と未知数の数はちょうど同じです．この場合の直線の方程式系を満たす交点は一つしかありません．図 8(b) には一つの直線があり，$\boldsymbol{A}$ は行フルランク（列フルランクではない）をもつ劣決定系となります．このとき，直線上のすべての点はこの直線の方程式を満たすので，それらの点はすべて交点となります．図 8(c) と (d) には三つの直線が存在します．$\boldsymbol{A}$ は列フルランク（行フルランクではない）をもっていて，方程式の数が未知数の数より多い過剰決定系となります．図 8(e) には二つの平行な直線があり，$\boldsymbol{A}$ はランク落ちの状態です．図 8(c, d, e) のそれぞれの直線の方程式を同時を満たす交点は存在せず，最適解しか求められません．

過剰決定線形システムの最適解　方程式系 $\boldsymbol{Ax} = \boldsymbol{b}$ が過剰決定となるとき，ある誤差を最小にする $\hat{\boldsymbol{x}}$ しか求められません．例えば，$\|\boldsymbol{b} - \boldsymbol{Ax}\|$ を誤差として設定すれば，以下の最小化問題となります．

$$\min \quad \|\boldsymbol{b} - \boldsymbol{Ax}\| \tag{33}$$

この問題は，以下のラグランジュの未定乗数法で解けます．

$$\mathcal{L}(\boldsymbol{x}) = \|\boldsymbol{b} - \boldsymbol{Ax}\| + \boldsymbol{\lambda}^\top 0 \quad \Rightarrow \quad \frac{\partial \mathcal{L}(\boldsymbol{x})}{\partial \boldsymbol{x}} = \frac{\partial \|\boldsymbol{b} - \boldsymbol{Ax}\|}{\partial \boldsymbol{x}} = 0 \tag{34}$$

すなわち，$\partial\|\boldsymbol{b}-\boldsymbol{Ax}\|/\partial\boldsymbol{x} = 0$ を解くことで，誤差 $\|\boldsymbol{b}-\boldsymbol{Ax}\|$ を最小にする $\hat{\boldsymbol{x}}$ が得られます．次に，$\partial\|\boldsymbol{b}-\boldsymbol{Ax}\|/\partial\boldsymbol{x} = 0$ の計算方法を紹介します．$\|\boldsymbol{b} - \boldsymbol{Ax}\|$ を展開すると，

$$\|\boldsymbol{b} - \boldsymbol{Ax}\| = (\boldsymbol{b} - \boldsymbol{Ax})^\top (\boldsymbol{b} - \boldsymbol{Ax}) = \boldsymbol{b}^\top \boldsymbol{b} - \boldsymbol{b}^\top \boldsymbol{Ax} - \boldsymbol{x}^\top \boldsymbol{A}^\top \boldsymbol{b} + \boldsymbol{x}^\top \boldsymbol{A}^\top \boldsymbol{Ax} \tag{35}$$

となります．この式の右辺を微分すると，以下の式のようになります．微分計算の詳しい説明については，本章の最後の補足資料に参考ください．

$$\frac{\partial(\boldsymbol{b}^\top \boldsymbol{b} - \boldsymbol{b}^\top \boldsymbol{Ax} - \boldsymbol{x}^\top \boldsymbol{A}^\top \boldsymbol{b} + \boldsymbol{x}^\top \boldsymbol{A}^\top \boldsymbol{Ax})}{\partial \boldsymbol{x}} = -2\boldsymbol{A}^\top \boldsymbol{b} + 2\boldsymbol{A}^\top \boldsymbol{Ax} \tag{36}$$

$\|\boldsymbol{b} - \boldsymbol{Ax}\|$ を最小にする解 $\hat{\boldsymbol{x}}$ は式 (36) が 0 になるときの $\boldsymbol{x}$ であるため，以下の方程式から得られます．

$$-2\boldsymbol{A}^\top \boldsymbol{b} + 2\boldsymbol{A}^\top \boldsymbol{Ax} = 0 \quad \Rightarrow \quad \boldsymbol{A}^\top \boldsymbol{Ax} = \boldsymbol{A}^\top \boldsymbol{b} \tag{37}$$

方程式系が過剰ですので，$\boldsymbol{A}$ は列フルランクを持ちます．そのため，$\boldsymbol{A}^\top\boldsymbol{A}$ の逆行列は存在します．これにより，$\hat{\boldsymbol{x}}$ は以下の式で計算できます．

$$\hat{\boldsymbol{x}} = (\boldsymbol{A}^\top\boldsymbol{A})^{-1}\boldsymbol{A}^\top\boldsymbol{b} \tag{38}$$

この式は $\|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}\|$ を最小化したときの解ですから，最小二乗法の解となります．図 8(c,d) に示した直線の方程式系は過剰決定ですので，式 (38) で最適な交点を計算できます．

$(\boldsymbol{A}^\top\boldsymbol{A})^{-1}\boldsymbol{A}^\top$ について，以下の特徴や別名を持っています． (1) 左逆行列：$(\boldsymbol{A}^\top\boldsymbol{A})^{-1}\boldsymbol{A}^\top$ を $\boldsymbol{A}$ の左側に掛けると，$(\boldsymbol{A}^\top\boldsymbol{A})^{-1}\boldsymbol{A}^\top\boldsymbol{A} = \boldsymbol{I}$ ですから，$(\boldsymbol{A}^\top\boldsymbol{A})^{-1}\boldsymbol{A}^\top$ は $\boldsymbol{A}$ の左逆行列とも呼ばれます．(2) 射影行列：ベクトル $\boldsymbol{b}$ が $\boldsymbol{x}$ を行列 $\boldsymbol{A}$ の列空間へ射影したものの観点から理解すると，射影後のベクトル $\boldsymbol{b} - \boldsymbol{A}\hat{\boldsymbol{x}}$ は行列 $\boldsymbol{A}$ の列空間と直交します．行列の列空間と直交するベクトルはこの行列の転置行列の零空間に存在するため，以下の式が成り立ちます．

$$\boldsymbol{A}^\top(\boldsymbol{b} - \boldsymbol{A}\hat{\boldsymbol{x}}) = 0 \tag{39}$$

この式を展開すると，同じ左逆行列が得られます．

$$\boldsymbol{A}^\top\boldsymbol{b} - \boldsymbol{A}^\top\boldsymbol{A}\hat{\boldsymbol{x}} = 0 \quad \Rightarrow \quad \hat{\boldsymbol{x}} = (\boldsymbol{A}^\top\boldsymbol{A})^{-1}\boldsymbol{A}^\top\boldsymbol{b} \tag{40}$$

右側の式の両辺に $\boldsymbol{A}$ を掛けると，以下の式が得られます．

$$\boldsymbol{A}\hat{\boldsymbol{x}} = \boldsymbol{A}(\boldsymbol{A}^\top\boldsymbol{A})^{-1}\boldsymbol{A}^\top\boldsymbol{b} \tag{41}$$

左辺の $\boldsymbol{A}\hat{\boldsymbol{x}}$ は行列 $\boldsymbol{A}$ の列空間にあるベクトルを意味します．右辺の $\boldsymbol{A}(\boldsymbol{A}^\top\boldsymbol{A})^{-1}\boldsymbol{A}^\top$ はベクトル $\boldsymbol{b}$ を $\boldsymbol{A}\hat{\boldsymbol{x}}$ に射影する行列を意味します．そのため，$\boldsymbol{A}(\boldsymbol{A}^\top\boldsymbol{A})^{-1}\boldsymbol{A}^\top$ は射影行列 (Projection Matrix) とも呼ばれます．

$\boldsymbol{A}$ がフルランクの（列フルランクをもつと同時に行フルランクをもつ）場合の解は過剰決定システムの解の一つの特殊なケースです。 $\boldsymbol{A}$ がフルランクのとき，列と行フルランクを同時に持ち，以下の計算が成り立ちます．

$$(\boldsymbol{A}^\top\boldsymbol{A})^{-1}\boldsymbol{A}^\top = \boldsymbol{A}^{-1}(\boldsymbol{A}^\top)^{-1}\boldsymbol{A}^\top = \boldsymbol{A}^{-1} \tag{42}$$

このことから，フルランクの解は列フルランクの場合の解の一つの特殊なケースであることがわかります。

リスト 4.2-4 は図 8(c) に示す直線の交点を計算するプログラムです．計算された交点は図 9(a.1) の青色のポイントです．

リスト 4.2-4: 図 8(c) に示す直線の交点を計算するプログラム

```
import numpy as np
import modeling.geometric_model as gm
import visualization.panda.world as wd
import basis.constant as cnst

if __name__ == '__main__':
    base = wd.World(cam_pos=np.array([.15, .1, 1]),
                    lookat_pos=np.array([.15, .1, 0]),
                    lens_type=wd.LensType.PERSPECTIVE)
    gm.gen_arrow(spos=np.zeros(3), epos=np.array([.3,0,0]),
                 rgba=cnst.black, stick_type="round").attach_to(base)
    gm.gen_arrow(spos=np.zeros(3), epos=np.array([0,.2,0]),
                 rgba=cnst.black, stick_type="round").attach_to(base)
    A = np.array([[-1, 2],
                  [.2, 1],
                  [.4, .2]])
    b = np.array([0.1, 0.1, 0.1])
    spos = np.zeros((3, 3))
    spos[:, 0] = 1
    spos[:, 1] = np.divide(b - A[:, 0], A[:, 1])
    epos = np.zeros((3, 3))
    epos[:, 0] = -1
    epos[:, 1] = np.divide(b + A[:, 0], A[:, 1])
    # 直線を画面に表示
    gm.gen_stick(spos[0, :], epos[0, :], rgba=cnst.magenta).attach_to(base)
    gm.gen_stick(spos[1, :], epos[1, :], rgba=cnst.yellow).attach_to(base)
    gm.gen_stick(spos[2, :], epos[2, :], rgba=cnst.cyan).attach_to(base)
    # 交点を計算する
```

```
    lsq_x = np.linalg.inv(A.T @ A) @ A.T @ b
    lsq_pos = np.zeros(3)
    lsq_pos[0:2] = lsq_x
    gm.gen_sphere(lsq_pos, radius=.0075, rgba=cnst.oriental_blue).attach_to(base)
    base.run()
```

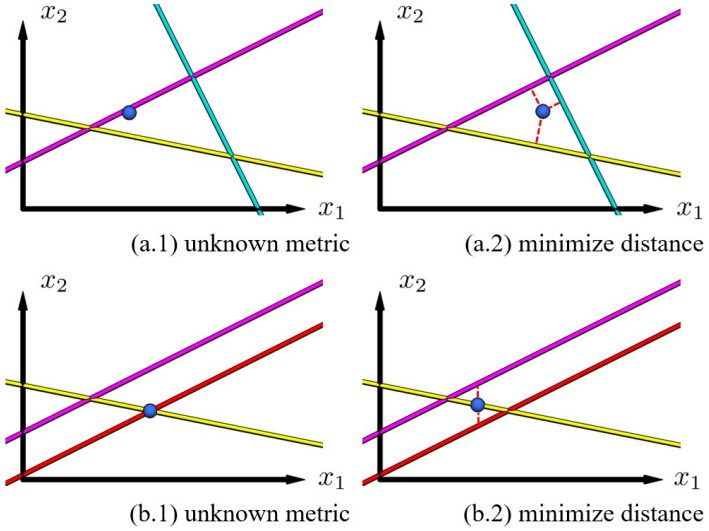

図 9: 最適化問題に異なる目的関数を設定した際の交点．(a) は図 8(c) の直線系に (a.1) $\|\boldsymbol{b}-\boldsymbol{A}\boldsymbol{x}\|$，(a.2) 直線までの距離の平方和 (式 (46)) を使用した場合． (b) は図 8(d) の直線系に (b.1) $\|\boldsymbol{b}-\boldsymbol{A}\boldsymbol{x}\|$，(b.2) 直線までの距離の平方和 (式 (46)) を使用した場合．

結果からみると，確かに式 (38) で交点を解けますが，解いた交点が何を最小化したのか，図 9(a.1) からはよくわかりません．これは，最適化の目的関数 $\|\boldsymbol{b}-\boldsymbol{A}\boldsymbol{x}\|$ の設定による問題です．$\|\boldsymbol{b}-\boldsymbol{A}\boldsymbol{x}\|$ の幾何的な意味が不明瞭なため，算出した交点の意味もはっきりしませんでした．一方，幾何的な意味をもつ最小化として，例えば，各直線までの距離の平方和の最小化を考える場合，まず点から直線までの距離の計算式から考えないといけません．

では，点から直線までの距離の計算式を考えましょう．点 (p_{x_1}, p_{x_2}) から直線 $a_1x_1+a_2x_2=b$ までの距離は以下の式で計算できます．

$$\frac{|a_1p_{x_1}+a_2p_{x_2}-b|}{\sqrt{{a_1}^2+{a_2}^2}} \tag{43}$$

すべての直線までの距離の平方和は以下のように表現できます．

$$\frac{(a_{11}p_{x_1}+a_{12}p_{x_2}-b_1)^2}{{a_{11}}^2+{a_{12}}^2}+\frac{(a_{21}p_{x_1}+a_{22}p_{x_2}-b_2)^2}{{a_{21}}^2+{a_{22}}^2}+\cdots$$

$$\Rightarrow \quad \left(\frac{a_{11}p_{x_1}+a_{12}p_{x_2}-b_1}{\sqrt{{a_{11}}^2+{a_{12}}^2}}\right)^2+\left(\frac{a_{21}p_{x_1}+a_{22}p_{x_2}-b_2}{\sqrt{{a_{21}}^2+{a_{22}}^2}}\right)^2+\cdots$$

$$\Rightarrow \quad \begin{bmatrix} \frac{a_{11}p_{x_1}+a_{12}p_{x_2}-b_1}{\sqrt{{a_{11}}^2+{a_{12}}^2}} & \frac{a_{21}p_{x_1}+a_{22}p_{x_2}-b_2}{\sqrt{{a_{21}}^2+{a_{22}}^2}} & \cdots \end{bmatrix} \begin{bmatrix} \frac{a_{11}p_{x_1}+a_{12}p_{x_2}-b_1}{\sqrt{{a_{11}}^2+{a_{12}}^2}} \\ \frac{a_{21}p_{x_1}+a_{22}p_{x_2}-b_2}{\sqrt{{a_{21}}^2+{a_{22}}^2}} \\ \vdots \end{bmatrix}$$

$$\Rightarrow \quad \left(\begin{bmatrix} \frac{1}{|\boldsymbol{A}_1|} & 0 & \cdots \\ 0 & \frac{1}{|\boldsymbol{A}_2|} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}\left(\boldsymbol{A}\begin{bmatrix}p_{x_1}\\p_{x_2}\end{bmatrix}-\boldsymbol{b}\right)\right)^\top \left(\begin{bmatrix} \frac{1}{|\boldsymbol{A}_1|} & 0 & \cdots \\ 0 & \frac{1}{|\boldsymbol{A}_2|} & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix}\left(\boldsymbol{A}\begin{bmatrix}p_{x_1}\\p_{x_2}\end{bmatrix}-\boldsymbol{b}\right)\right)$$

$$\Rightarrow \quad \left(\mathrm{diag}(|\boldsymbol{A}_1|,|\boldsymbol{A}_2|,\cdots)^{-1}(\boldsymbol{A}\begin{bmatrix}p_{x_1}\\p_{x_2}\end{bmatrix}-\boldsymbol{b})\right)^{\top}\left(\mathrm{diag}(|\boldsymbol{A}_1|,|\boldsymbol{A}_2|,\cdots)^{-1}(\boldsymbol{A}\begin{bmatrix}p_{x_1}\\p_{x_2}\end{bmatrix}-\boldsymbol{b})\right) \tag{44}$$

式 (44) 中の $\boldsymbol{A}_i$ は行列 $\boldsymbol{A}$ の i 行を指し，$1/|\boldsymbol{A}_i|$ は以下のように計算されます．

$$\frac{1}{|\boldsymbol{A}_i|}=\frac{1}{\sqrt{{a_{i1}}^2+{a_{i2}}^2}} \tag{45}$$

また，diag() は括弧内のベクトルの各要素を対角成分とする対角行列を作る関数です．式 (44) により，各直線までの距離の平方和を最小にする交点は，以下の式を最小にする $\hat{\boldsymbol{x}}$ となります．

$$\begin{aligned}&\left(\mathrm{diag}(|\boldsymbol{A}_1|,|\boldsymbol{A}_2|,\cdots)^{-1}(\boldsymbol{A}\boldsymbol{x}-\boldsymbol{b})\right)^{\top}\left(\mathrm{diag}(|\boldsymbol{A}_1|,|\boldsymbol{A}_2|,\cdots)^{-1}(\boldsymbol{A}\boldsymbol{x}-\boldsymbol{b})\right)\\ \Rightarrow\quad &(\boldsymbol{A}'\boldsymbol{x}-\boldsymbol{b}')^{\top}(\boldsymbol{A}'\boldsymbol{x}-\boldsymbol{b}')=\|\boldsymbol{b}'-\boldsymbol{A}'\boldsymbol{x}\|\end{aligned} \tag{46}$$

ここで，$\boldsymbol{A}'$ と $\boldsymbol{b}'$ は以下のようにおいています．

$$\boldsymbol{A}'=\left(\mathrm{diag}(|\boldsymbol{A}_1|,|\boldsymbol{A}_2|,\cdots)^{-1}\boldsymbol{A}\right),\quad \boldsymbol{b}'=\left(\mathrm{diag}(|\boldsymbol{A}_1|,|\boldsymbol{A}_2|,\cdots)^{-1}\boldsymbol{b}\right) \tag{47}$$

式 (46) により，各直線までの距離の平方和を最小にする交点は，以下の方程式が成り立つときの $\hat{\boldsymbol{x}}$ であることがわかります．

$$\frac{\partial\|\boldsymbol{b}'-\boldsymbol{A}'\boldsymbol{x}\|}{\partial\boldsymbol{x}}=0 \tag{48}$$

式 (35)〜(38) と同じ流れで導出すると，

$$\hat{\boldsymbol{x}}=({\boldsymbol{A}'}^{\top}\boldsymbol{A}')^{-1}{\boldsymbol{A}'}^{\top}\boldsymbol{b}' \tag{49}$$

となります．これにしたがって，リスト 4.2-4 の 26 行で使われる $\boldsymbol{A}$ と $\boldsymbol{b}$ を $\boldsymbol{A}'$ と $\boldsymbol{b}'$ で置き換えれば，各直線までの距離の平方和を最小にする $\hat{\boldsymbol{x}}$ も計算できるでしょう．

以下のコードは $\boldsymbol{A}'$ と $\boldsymbol{b}'$ の計算を実装したものです．これらのコードをリスト 4.2-4 の 26 行の前に挿入すると，図 9(c) の結果が得られます．

```
1     A_norm = np.linalg.norm(A, axis=1)
2     A_tf = np.linalg.inv(np.\textnormal{diag}(A_norm))
3     A = A_tf @ A
4     b = A_tf @ b
```

図 8(d) に示す直線の交点を求める計算も同じ流れで実装できます．図 9(b.1)，(b.2) は，最適化の目的関数をそれぞれ $\|\boldsymbol{b}-\boldsymbol{A}\boldsymbol{x}\|$，$\|\boldsymbol{b}'-\boldsymbol{A}'\boldsymbol{x}\|$ に設定したときの交点を示しています．

劣決定線形システムの最適解　方程式系 $\boldsymbol{A}\boldsymbol{x}=\boldsymbol{b}$ が劣決定であるとき，$\boldsymbol{A}\boldsymbol{x}=\boldsymbol{b}$ を満たす $\boldsymbol{x}$ は無限に存在します．その場合，$\boldsymbol{A}$ は列フルランクをもっておらず，行フルランクをもっています．そのため $\boldsymbol{A}^{\top}\boldsymbol{A}$ は存在しません．式 (38) は成り立ちません．$\|\boldsymbol{A}\boldsymbol{x}-\boldsymbol{b}\|$ の最小値は 0，その最小値を与える $\boldsymbol{x}$ は無限に存在しますので，式 (33) に示す最小化問題も解けません．そのため，劣決定線形システムの最適解問題の設定を改めて考えます．複数の解から最適な値を求める必要ですので，$\boldsymbol{A}\boldsymbol{x}=\boldsymbol{b}$ を拘束条件として捉え，この拘束条件を満たす $\boldsymbol{x}$ の中から指定された目標を最小にするものを求める問題として考えます．指定された目標は $\boldsymbol{x}^{\top}\boldsymbol{x}$ の時，新たな最適化問題は下記の式になります．

$$\begin{aligned}\min\quad &\|\boldsymbol{x}\|\\ \text{s.t.}\quad &\boldsymbol{A}\boldsymbol{x}=\boldsymbol{b}\end{aligned} \tag{50}$$

この問題は，式 (33) に示される問題と同様にラグランジュの未定乗数法で解けます．

$$\begin{aligned}&\mathcal{L}(\boldsymbol{x})=\|\boldsymbol{x}\|+\boldsymbol{\lambda}^{\top}(\boldsymbol{A}\boldsymbol{x}-\boldsymbol{b})=\boldsymbol{x}^{\top}\boldsymbol{x}+\boldsymbol{\lambda}^{\top}(\boldsymbol{A}\boldsymbol{x}-\boldsymbol{b})\\ \Rightarrow\quad &\frac{\partial\mathcal{L}(\boldsymbol{x})}{\partial\boldsymbol{x}}=\frac{\partial(\boldsymbol{x}^{\top}\boldsymbol{x}+\boldsymbol{\lambda}^{\top}(\boldsymbol{A}\boldsymbol{x}-\boldsymbol{b}))}{\partial\boldsymbol{x}}=2\boldsymbol{x}+\boldsymbol{A}^{\top}\boldsymbol{\lambda}=0\end{aligned} \tag{51}$$

拘束条件 $\boldsymbol{Ax} = \boldsymbol{b}$ とラグランジュの未定乗数法の式を合わせて解くと，$\boldsymbol{\lambda}$ と $\hat{\boldsymbol{x}}$ の値はそれぞれ以下の式になります．これらの式を導出するには，$\boldsymbol{A}$ が行フルランクをもち，$\boldsymbol{AA}^\top$ の逆行列が常に存在することを利用しています．

$$\begin{cases} 2\boldsymbol{x} + \boldsymbol{A}^\top\boldsymbol{\lambda} = 0 \\ \boldsymbol{Ax} = \boldsymbol{b} \end{cases} \quad \Rightarrow \quad \begin{cases} \boldsymbol{\lambda} = -2(\boldsymbol{AA}^\top)^{-1}\boldsymbol{b} \\ \hat{\boldsymbol{x}} = \boldsymbol{A}^\top(\boldsymbol{AA}^\top)^{-1}\boldsymbol{b} \end{cases} \tag{52}$$

ここで注意してもらいたいことは，式 (50) は過剰決定システムに適用できないということです．線形システムの方程式の数が未知数の数より多い場合，拘束条件 $\boldsymbol{Ax} = \boldsymbol{b}$ を満たす解は存在しないため，式 (50) に示す最小化問題は成り立ちません．また，$\boldsymbol{A}$ がフルランクをもつ（列フルランクをもつと同時に行フルランクをもつ）場合の解は，式 (42) の導出と同じように，劣決定線形システムの最適解の特殊な形式です．$\boldsymbol{A}$ はフルランクですので，以下の計算が成り立ちます．

$$\boldsymbol{A}^\top(\boldsymbol{AA}^\top)^{-1} = \boldsymbol{A}^\top(\boldsymbol{A}^\top)^{-1}\boldsymbol{A}^{-1} = \boldsymbol{A}^{-1} \tag{53}$$

そのため，$\boldsymbol{A}$ がフルランクをもつ場合 $\hat{\boldsymbol{x}} = \boldsymbol{A}^\top(\boldsymbol{AA}^\top)^{-1}\boldsymbol{b} = \boldsymbol{A}^{-1}\boldsymbol{b}$ となります．

リスト 4.2-5 は式 (52) を利用した図 8(b) に示された交点問題の実装です．結果は図 11 に示します．$\|\boldsymbol{x}\|$ を最小化していますから，求めた交点は原点から直線まで下ろした垂線の足となります．

リスト 4.2-5: 図 8(b) に示す直線の “ 交点 ” を計算するプログラム

```
    # importや座標系などのレンダリングを省略しています...
    A = np.array([[-1, 2]])
    b = np.array([0.1])
    spos = np.zeros((1, 3))
    spos[:, 0] = 1
    spos[:, 1] = np.divide(b - A[:, 0], A[:, 1])
    epos = np.zeros((1, 3))
    epos[:, 0] = -1
    epos[:, 1] = np.divide(b + A[:, 0], A[:, 1])
    # 直線を画面に表示
    gm.gen_stick(spos[0, :], epos[0, :],
                 rgba=cnst.magenta).attach_to(base)
    # 交点を計算する
    lsq_x = A.T @ np.linalg.inv(A @ A.T) @ b
    lsq_pos = np.zeros(3)
    lsq_pos[:2] = lsq_x
    gm.gen_sphere(lsq_pos, radius=.0075,
                  rgba=cnst.oriental_blue).attach_to(base)
    base.run()
```

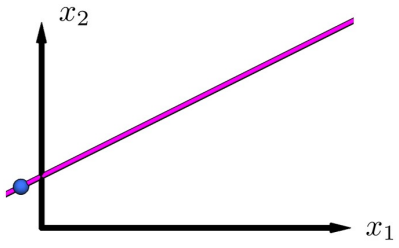

図 10: 図 8(b) に示す直線の “ 交点 ”

任意線形システムの最適解　線形システムの決定性がわからない場合，どのように最適解を求めるか考えましょう．このとき，線形システムは過剰決定と劣決定の両方の可能性があります．また，どちらの決定性にもならない可能性もあります．図 8(e) に示すランク落ちの直線系は過剰決定でも劣決定でもないシステムの例です．この例において，$\boldsymbol{Ax} = \boldsymbol{b}$ を満たす $\boldsymbol{x}$ の厳密解は存在しません．同時に，$\|\boldsymbol{b} - \boldsymbol{Ax}\|$ の最小値を与える $\boldsymbol{x}$ は無限個存在します．

以上のことから，線形システムの決定性がわからないとき，式 (33) と式 (50) のどちらも不成立となる可能性があります．これにより，式 (33) と式 (50) を結合した以下の問題を解くことで，任意線形システムの最適解を求めます．

$$\begin{aligned} \min \quad & \|\boldsymbol{b} - \boldsymbol{Ax}\| + \lambda\|\boldsymbol{x}\| \\ \text{s.t.} \quad & \text{なし} \end{aligned} \tag{54}$$

この式の λ は二番目の項影響をコントロールするためのパラメータです．線形システムが過剰決定のとき，この問題は $\|\boldsymbol{b} - \boldsymbol{Ax}\|$ を最小にするための問題として扱われるため，λ が 0 に近づくほど問題の設定が正しくなります．一方，線形システムが劣決定のとき，$\|\boldsymbol{b} - \boldsymbol{Ax}\|$ は常に 0 になります．この場合，この問題を $\lambda\|\boldsymbol{x}\|$ の最小化問題として扱うため，λ を 0 に設定すべきではありません．また，どちらの決定性でもない場合，$\|\boldsymbol{b} - \boldsymbol{Ax}\|$ を最小にすると同時に $\lambda\|\boldsymbol{x}\|$ も最小にする問題として扱うため，$\lambda\|\boldsymbol{x}\|$ の影響を弱く設定したほうがよいことがわかります．これらのことから，λ を 0 とみなされない範囲で小さい正の値に設定することが理想的です．

式 (54) は式 (33)，式 (50) と同様にラグランジュの未定乗数法で解けます．

$$\begin{aligned} & \mathcal{L}(\boldsymbol{x}) = \|\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}\| + \lambda\|\boldsymbol{x}\| = (\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x})^\top(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}) + \lambda\boldsymbol{x}^\top\boldsymbol{x} \\ \Rightarrow \quad & \frac{\partial\mathcal{L}(\boldsymbol{x})}{\partial\boldsymbol{x}} = \frac{\partial((\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x})^\top(\boldsymbol{b} - \boldsymbol{A}\boldsymbol{x}) + \lambda\boldsymbol{x}^\top\boldsymbol{x})}{\partial\boldsymbol{x}} = 0 \\ \Rightarrow \quad & -2\boldsymbol{A}^\top\boldsymbol{b} + 2\boldsymbol{A}^\top\boldsymbol{A}\boldsymbol{x} + 2\lambda\boldsymbol{I}\boldsymbol{x} = 0 \\ \Rightarrow \quad & (\boldsymbol{A}^\top\boldsymbol{A} + \lambda\boldsymbol{I})\boldsymbol{x} = \boldsymbol{A}^\top\boldsymbol{b} \end{aligned} \tag{55}$$

λ は 0 にならない小さい正の値なので，上式の左辺の $(\boldsymbol{A}^\top\boldsymbol{A} + \lambda\boldsymbol{I})$ の逆行列は常に存在することがわかります．これにより，任意線形システムの最適解は以下の式で計算できます．

$$\hat{\boldsymbol{x}} = (\boldsymbol{A}^\top\boldsymbol{A} + \lambda\boldsymbol{I})^{-1}\boldsymbol{A}^\top\boldsymbol{b} \tag{56}$$

この結果を用いて，図 8(e) に示した交点問題も解けます．リスト 4.2-6 は実装です．結果は図 11 に示します．$\|\boldsymbol{x}\|$ を最小化していますから，求めた交点は原点から直線まで下ろした垂線の足となります．

リスト 4.2-6: 図 8(e) に示す直線の交点を計算するプログラム

```
    # importや座標系などのレンダリングを省略しています...
    A = np.array([[-1, 2],
                  [-10, 20]])
    b = np.array([0.1, 0.1])
    spos = np.zeros((2, 3))
    spos[:, 0] = 1
    spos[:, 1] = np.divide(b - A[:, 0], A[:, 1])
    epos = np.zeros((2, 3))
    epos[:, 0] = -1
    epos[:, 1] = np.divide(b + A[:, 0], A[:, 1])
    # 直線を画面に表示
    gm.gen_stick(spos[0, :], epos[0, :],
                 rgba=cnst.magenta).attach_to(base)
    gm.gen_stick(spos[1, :], epos[1, :],
                 rgba=cnst.red).attach_to(base)
    # 交点を計算する
    damped_inv = np.linalg.inv(A.T @ A+1e-4*np.eye(A.shape
      [0]))
    lsq_x = damped_inv @ A.T @ b
    lsq_x1 = np.linalg.pinv(A) @ b
    print(lsq_x, lsq_x1)
    lsq_pos = np.zeros(3)
    lsq_pos[:2] = lsq_x
    gm.gen_sphere(lsq_pos, radius=.0075,
                  rgba=cnst.oriental_blue).attach_to(base)
    base.run()
```

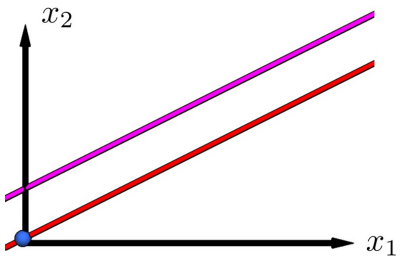

図 11: 図 8(e) に示す直線の交点

減衰最小二乗法と逆運動学への適用　上の議論から，λ を非常に小さい値に設定するか動的に調整することで，式 (56) が線形方程式系のさまざまな状況を網羅できることがわかります。したがって，この方法を用いて修正量を求め，数値的な逆運動学の解を導くことが可能です。最小二乗法の $(\boldsymbol{A}^\top\boldsymbol{A})^{-1}\boldsymbol{A}^\top$ と比べると，式 (56) の係数部分の $(\boldsymbol{A}^\top\boldsymbol{A} + \lambda\boldsymbol{I})^{-1}\boldsymbol{A}^\top$ は対角行列 $\lambda\boldsymbol{I}$ を $\boldsymbol{A}^\top\boldsymbol{A}$ に加えることで逆行列が存在することを保障しています．λ は行列ゼロをならないような減衰効果を持つので，減衰パラメータと呼ばれ，λ の動的な調整によって残差を効果的に制御することも可能です．このため，式 (56) に示された手法は減衰最小二乗法（Damped Least Squre, DLS）と呼ばれます．減衰最小二乗法はアメリカの統計学者レーベンバーグとマルカートにより提案されましたので，レーベンバーグ・マルカート法（Levenberg-Marquardt algorithm, LM 法）とも呼ばれます．

ニュートン-ラフソン法の繰り返しごとの修正の目標は，$[\delta\boldsymbol{p}(\delta\boldsymbol{q}_i)\ \delta\boldsymbol{\xi}(\delta\boldsymbol{q}_i]^\top = \boldsymbol{J}(\boldsymbol{q}_i)\delta\boldsymbol{q}_i$ に示される線形システムに満たす $\delta\boldsymbol{q}_i$ を求めることです．ここで，$\boldsymbol{J}(\boldsymbol{q}_i)$ は前述した線型方程式系の行列係数 $\boldsymbol{A}$，$\delta\boldsymbol{q}_i$ は $\boldsymbol{x}$，$[\delta\boldsymbol{p}(\delta\boldsymbol{q}_i)\ \delta\boldsymbol{\xi}(\delta\boldsymbol{q}_i]^\top$ は $\boldsymbol{b}$ に相当します．$\boldsymbol{J}(\boldsymbol{q}_i)$ の決定形態は，直接リンクの関節の数と回転角度によって異なる場合があるため，減衰最小二乗法を使用して解を求めるべきです。計算式は下記の通りです。

$$\delta\boldsymbol{q}_i = (\boldsymbol{J}(\boldsymbol{q}_i)^\top\boldsymbol{J}(\boldsymbol{q}_i) + \lambda\boldsymbol{I})^{-1}\boldsymbol{J}(\boldsymbol{q}_i)^\top \begin{bmatrix} \delta\boldsymbol{p}(\delta\boldsymbol{q}_i) \\ \delta\boldsymbol{\xi}(\delta\boldsymbol{q}_i) \end{bmatrix} \tag{57}$$

この式を用いて，ニュートン-ラフソン法の修正量を計算し，繰り返し更新すれば，逆運動学を数値的に解けるでしょう．

2.2.4 関節角度の修正量の求解 2：疑似逆行列法

特異値分解による疑似逆行列法を導出 ここまで議論した線形システムの最適化解法以外に，疑似逆行列で修正量を解く方法もあります．これから，疑似逆行列を使った手法について説明します．疑似逆行列は一般に，アメリカの数学者ムーアとイギリスの物理学者ペンローズなどにより提案されたムーア・ペンローズ逆行列を指します．疑似逆行列は特異値分解を用いて計算できます．本資料では皆さんが特異値分解の背景知識をもたないときを想定しますから，まず，特異値分解の背景知識を簡単に紹介します．特異値分解 (Singular Value Decomposition, SVD) は固有値分解を一般化したものです．行列 $\boldsymbol{A}$ が正則（可逆）の場合，以下のように対角化と固有値分解できること，皆さんすでにご存知だと思います．ここで $\boldsymbol{P}$ は固有ベクトルを並べた行列です．対角行列 $\boldsymbol{\Lambda}$ の対角成分は $\boldsymbol{A}$ の固有値です．

$$\boldsymbol{A} = \boldsymbol{P}\boldsymbol{\Lambda}\boldsymbol{P}^{-1} \tag{58}$$

$\boldsymbol{A}$ が非正則行列の場合でも，類似の方法で分解することができます。例えば，$\boldsymbol{A}$ が $m \times n$ の行列でランク $r \leq \min(m, n)$ をもつ場合，以下の式に示す特異値と特異ベクトルの関係をもっています．

$$\boldsymbol{A}\boldsymbol{v}_1 = \sigma_1\boldsymbol{u}_1,\ \boldsymbol{A}\boldsymbol{v}_2 = \sigma_2\boldsymbol{u}_2,\ \cdots,\ \boldsymbol{A}\boldsymbol{v}_r = \sigma_1\boldsymbol{u}_r;\quad \boldsymbol{A}\boldsymbol{v}_{r+1} = 0,\ \boldsymbol{A}\boldsymbol{v}_{r+2} = 0,\ \cdots,\ \boldsymbol{A}\boldsymbol{v}_m = 0 \tag{59}$$

ここで，$\boldsymbol{v}_1, \boldsymbol{v}_2, \cdots, \boldsymbol{v}_m$ は行列 $\boldsymbol{A}$ の右特異ベクトルと呼ばれ，$\boldsymbol{A}$ の行空間の基底を示します．$\boldsymbol{u}_1, \boldsymbol{u}_2, \cdots, \boldsymbol{u}_n$ は行列 $\boldsymbol{A}$ の左特異ベクトルと呼ばれ，$\boldsymbol{A}$ の列空間の基底を示します．特異値と特異ベクトルの概念は固有値と固有ベクトルの定義と類似しています．任意の行列 $\boldsymbol{A}$ に写像された後のベクトルが写像される前のベクトルのスカラー倍になるとき，このスカラーとベクトルはそれぞれ特異値と特異ベクトルと呼ばれます．固有値と固有ベクトルと比べると，特異値と特異ベクトルは正則行列に縛られる概念の代わり，任意の行列に適用できる一般化した概念です．式 (59) を整理すると，正則行列の対角化と似たような式を貰えます．

$$\boldsymbol{A}\begin{bmatrix} | & | & & | & & | \\ \boldsymbol{v}_1 & \boldsymbol{v}_2 & \cdots & \boldsymbol{v}_r & \cdots & \boldsymbol{v}_m \\ | & | & & | & & | \end{bmatrix} = \begin{bmatrix} | & | & & | & & | \\ \boldsymbol{u}_1 & \boldsymbol{u}_2 & \cdots & \boldsymbol{u}_r & \cdots & \boldsymbol{u}_n \\ | & | & & | & & | \end{bmatrix} \left[\begin{array}{ccc|c} \sigma_1 & & & & \\ & \sigma_2 & & & \\ & & \ddots & & 0 \\ & & & \sigma_r & \\ \hline & 0 & & & 0 \end{array}\right]$$

$$\Rightarrow \quad \boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top \tag{60}$$

この式の $\boldsymbol{U}$ と $\boldsymbol{V}^\top$ はそれぞれ，

$$\boldsymbol{U} = \begin{bmatrix} | & | & & | & & | \\ \boldsymbol{u}_1 & \boldsymbol{u}_2 & \cdots & \boldsymbol{u}_r & \cdots & \boldsymbol{u}_n \\ | & | & & | & & | \end{bmatrix},\quad \boldsymbol{V}^\top = \begin{bmatrix} — & \boldsymbol{v}_1 & — \\ — & \boldsymbol{v}_2 & — \\ & \vdots & \\ — & \boldsymbol{v}_m & — \end{bmatrix} \tag{61}$$

です．式 (60) はしばしば以下のように書かれます．

$$\boldsymbol{A} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top = \sum_{i=1}^{r} \sigma_i \boldsymbol{u}_i \boldsymbol{v}_i^\top \tag{62}$$

式 (60) は任意の行列 $\boldsymbol{A}$ の特異値分解の公式です．$\boldsymbol{A}$ が正則のとき，式 (62) は固有値分解の公式と同じ形になります．

特異値分解を用いれば，行列 $\boldsymbol{A}$ の疑似逆行列を簡単に計算できます．式 $\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b}$ を考えましょう．$\boldsymbol{A}$ を特異値分解の式 (60) で置き換えると，式 $\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top\boldsymbol{x} = \boldsymbol{b}$ が得られます．つまり，$\boldsymbol{x}$ と $\boldsymbol{b}$ には以下に示される線形変換関係があります．

$$\boldsymbol{x} \xrightarrow{\text{rotate}} \boldsymbol{V}^\top\boldsymbol{x} \xrightarrow{\text{stretch}} \boldsymbol{\Sigma}\boldsymbol{V}^\top\boldsymbol{x} \xrightarrow{\text{rotate}} \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top\boldsymbol{x} = \boldsymbol{b} \tag{63}$$

逆に計算すれば,

$$\boldsymbol{b} = \boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top\boldsymbol{x} \xrightarrow{\text{unrotate}} \boldsymbol{U}^\top(\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top\boldsymbol{x}) = \boldsymbol{\Sigma}\boldsymbol{V}^\top\boldsymbol{x} \xrightarrow{\text{unstretch}} \boldsymbol{\Sigma}^{-1}(\boldsymbol{\Sigma}\boldsymbol{V}^\top\boldsymbol{x}) = \boldsymbol{V}^\top\boldsymbol{x} \xrightarrow{\text{unrotate}} \boldsymbol{V}(\boldsymbol{V}^\top\boldsymbol{x}) = \boldsymbol{x} \quad (64)$$

となります．ここで，$\boldsymbol{U}$ と $\boldsymbol{V}$ の各列は $\boldsymbol{A}$ の列空間と行空間の基底ですから，$\boldsymbol{U}^\top\boldsymbol{U} = \boldsymbol{I}$ と $\boldsymbol{V}\boldsymbol{V}^\top = \boldsymbol{I}$ の二つの性質を利用しています．この式により，$\boldsymbol{V}\boldsymbol{\Sigma}^{-1}\boldsymbol{U}^\top\boldsymbol{b} = \boldsymbol{V}\boldsymbol{\Sigma}^{-1}\boldsymbol{U}^\top\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top\boldsymbol{x}=\boldsymbol{x}$ が分かり，$\boldsymbol{V}\boldsymbol{\Sigma}^{-1}\boldsymbol{U}^\top$ は $\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$ の逆行列となります.

ただ，$\boldsymbol{\Sigma}^{-1}$ の計算には問題が残っています．$\boldsymbol{\Sigma}$ は特異ベクトル方向にどれだけストレッチ（伸縮）するかを表します。特異値が大きいほど，その方向へのストレッチが強く，小さい特異値はその方向へのストレッチが弱いことを示します。$\boldsymbol{\Sigma}$ には特異値がゼロの要素があります．特異値がゼロであることは，その方向には影響を与えないことを示しています。ゼロの逆数は無限大ですから，上式で解いた行列 $\boldsymbol{V}\boldsymbol{\Sigma}^{-1}\boldsymbol{U}^\top$ にも無限大の要素を含めます．そのような逆行列は正しくではありません.

正しく逆行列を解くため，改めて式 (64) の計算過程を確認してみます．$\boldsymbol{\Sigma}^{-1}$ の計算において，特異値がゼロである場合，その逆数は式 (64) の計算中にゼロに掛けますから，ゼロのただの逆数（無限大）でも，任意の値に設定しても計算結果に実質的な影響はありません。ところで，計算の簡便化と数値的な安定性を考慮して，特異値がゼロの場合はその逆数をゼロに設定します。そうすると，$\boldsymbol{\Sigma}^{-1}$ は以下の式のように 0 ではない対角項の逆数のみ計算します.

$$\boldsymbol{\Sigma}^{-1} = \left[\begin{array}{cccc|c} \sigma_1^{-1} & & & & \\ & \sigma_2^{-1} & & & \\ & & \ddots & & 0 \\ & & & \sigma_r^{-1} & \\ \hline & 0 & & & 0 \end{array}\right] \quad (65)$$

このように計算した $\boldsymbol{V}\boldsymbol{\Sigma}^{-1}\boldsymbol{U}^\top$ は行列 $\boldsymbol{A}$ の疑似逆行列と呼ばれます．疑似逆行列の $\boldsymbol{\Sigma}^{-1}$ の逆計算は 0 以外の要素のみに対して実施していますので，$\boldsymbol{\Sigma}^\dagger$ とも記されます．これにより，疑似逆行列の算式は下のように書かれたこともよく見えます.

$$\boldsymbol{A}^\dagger = \boldsymbol{V}\boldsymbol{\Sigma}^\dagger\boldsymbol{U}^\top. \quad (66)$$

では，疑似逆行列を用いて図 8(c)∼(e) の交点を計算してみましょう．リスト 4.2-4 の 26 行を以下のコードに置き換えるだけで，疑似逆行列を用いた計算を簡単に実装できます．結果は図 9(a.1)，(b.1) と同じです．これにより，疑似逆行列の結果は最適化による手法と類似することがわかります.

```
1    lsq_x = np.linalg.pinv(A) @ b
```

疑似逆行列法と減衰最小二乗法の関係　疑似逆行列法と減衰最小二乗法両方とも行列の逆問題において，異なる行列のランクに対応できる一般的な方法です。両方法の行列を並べてみると，下の式になります.

$$\begin{cases} \text{疑似逆行列: } \boldsymbol{V}\boldsymbol{\Sigma}^\dagger\boldsymbol{U}^\top \\ \text{減衰最小二乗行列: } (\boldsymbol{A}^\top\boldsymbol{A} + \lambda\boldsymbol{I})^{-1}\boldsymbol{A}^\top \end{cases} \quad (67)$$

比較しにくいため，二番目の式の $\boldsymbol{A}$ を $\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top$ で代替してみます.

$$\begin{aligned} (\boldsymbol{A}^\top\boldsymbol{A} + \lambda\boldsymbol{I})^{-1}\boldsymbol{A}^\top &= ((\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top)^\top(\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top) + \lambda\boldsymbol{I})^{-1}(\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top)^\top \\ &= ((\boldsymbol{V}\boldsymbol{\Sigma}^\top\boldsymbol{U}^\top)(\boldsymbol{U}\boldsymbol{\Sigma}\boldsymbol{V}^\top) + \lambda\boldsymbol{I})^{-1}(\boldsymbol{V}\boldsymbol{\Sigma}^\top\boldsymbol{U}^\top) \\ &= (\boldsymbol{V}\boldsymbol{\Sigma}^\top\boldsymbol{\Sigma}\boldsymbol{V}^\top + \lambda\boldsymbol{I})^{-1}(\boldsymbol{V}\boldsymbol{\Sigma}^\top\boldsymbol{U}^\top) \\ &= (\boldsymbol{V}\boldsymbol{\Sigma}^\top\boldsymbol{\Sigma}\boldsymbol{V}^\top + \boldsymbol{V}\lambda\boldsymbol{I}\boldsymbol{V}^\top)^{-1}(\boldsymbol{V}\boldsymbol{\Sigma}^\top\boldsymbol{U}^\top) \\ &= \underline{(\boldsymbol{V}(\boldsymbol{\Sigma}^\top\boldsymbol{\Sigma} + \lambda\boldsymbol{I})\boldsymbol{V}^\top)^{-1}}(\boldsymbol{V}\boldsymbol{\Sigma}^\top\boldsymbol{U}^\top) \end{aligned} \quad (68)$$

下線の部分 $(\boldsymbol{V}(\boldsymbol{\Sigma}^\top\boldsymbol{\Sigma} + \lambda\boldsymbol{I})\boldsymbol{V}^\top)^{-1}$ は逆計算ですので，式 (64) に従って，$(\boldsymbol{V}(\boldsymbol{\Sigma}^\top\boldsymbol{\Sigma} + \lambda\boldsymbol{I})\boldsymbol{V}^\top)^{-1} = \boldsymbol{V}(\boldsymbol{\Sigma}^\top\boldsymbol{\Sigma} + \lambda\boldsymbol{I})^{-1}\boldsymbol{V}^\top$ が分かります．続いて,

$$(\boldsymbol{V}(\boldsymbol{\Sigma}^\top\boldsymbol{\Sigma} + \lambda\boldsymbol{I})\boldsymbol{V}^\top)^{-1}(\boldsymbol{V}\boldsymbol{\Sigma}^\top\boldsymbol{U}^\top) = \boldsymbol{V}(\boldsymbol{\Sigma}^\top\boldsymbol{\Sigma} + \lambda\boldsymbol{I})^{-1}\boldsymbol{V}^\top\boldsymbol{V}\boldsymbol{\Sigma}^\top\boldsymbol{U}^\top$$

$$
\begin{aligned}
&= \boldsymbol{V}((\boldsymbol{\Sigma}^\top\boldsymbol{\Sigma} + \lambda\boldsymbol{I})^{-1}\boldsymbol{\Sigma}^\top)\boldsymbol{U}^\top \\
&= \sum_{i=1}^{r} \frac{\sigma_i}{\sigma_i^2 + \lambda}\boldsymbol{u}_i\boldsymbol{v}_i^\top
\end{aligned} \tag{69}
$$

が分かります．総合和の形式で，も一度疑似逆行列と減衰最小二乗行列を並べてみると，擬似逆行列法と減衰最小二乗法の関係が明確になります。

$$
\begin{cases}
\text{疑似逆行列:} \displaystyle\sum_{i=1}^{r} \frac{1}{\sigma_i}\boldsymbol{u}_i\boldsymbol{v}_i^\top \\
\text{減衰最小二乗行列:} \displaystyle\sum_{i=1}^{r} \frac{\sigma_i}{\sigma_i^2 + \lambda}\boldsymbol{u}_i\boldsymbol{v}_i^\top
\end{cases} \tag{70}
$$

どちらの場合もの式 $\sum_{i=1}^{r} \square\boldsymbol{u}_i\boldsymbol{v}_i^\top$ によって「逆行列化」されます。擬似逆行列の場合，$\square$ は ${\sigma_i}^{-1}$ です．この場合，$0^{-1} = 0$ と設定します．減衰最小二乗法の場合，$\square$ は $\sigma_i/(\sigma_i^2 + \lambda)$ です．λ は正の小さい値ですから，σ_i は 0 になっても $\sigma_i/(\sigma_i^2 + \lambda)$ は 0^{-1} になりません．σ_i の値が大きい場合，減衰最小二乗法は擬似逆行列法とほとんど違いがありません。そのとき，$\sigma_i/(\sigma_i^2 + \lambda) \approx {\sigma_i}^{-1}$．しかし，$\sigma_i$ がゼロに近づくほど，減衰最小二乗法は擬似逆行列法と違います．減衰最小二乗法は安定かつ滑らかに 0 になることに対して，擬似逆行列法で計算した逆行列は急に大きくなって，不安定になります．この問題を解決するために，擬似逆行列を求める際に単に 0^{-1} を 0 を設定するだけでなく，一定の値以下の σ_i の逆数をすべて 0 にするように切り上げることは有効です。 NumPy の linalg.pinv 関数は，この切り上げる機能を実現するための rcond 引数を提供しています。この引数に大きな値を調整することで，非常に小さい σ_i による巨大な逆数の影響を除去し，擬似逆行列の安定性を確保し，数値逆解を成功裏に求めることができます。

2.2.5 WRS におけるの実装

一般的な実装 WRS のニュートン-ラフソン法による逆運動学計算には，減衰最小二乗法の実装と疑似逆行列法の実装の両方が含まれています。それは，robot_sim/_kinematics/ik_num の NumIKSolver クラスの dls で始まるメンバー関数と pinv で始まるメンバー変数を参考にしてください。その中でも特に，pinv() 関数の中身は以下の通りで，逆疑似逆行列法を実装しています。

リスト 4.2-7: 減衰最小二乗法の実装

```
    def pinv(self, tgt_pos, tgt_rotmat, seed_jnt_values=None,
             max_n_iter=100, toggle_dbg=False):
        """
        :param tgt_pos: 目標の位置
        :param tgt_rotmat: 目標の回転行列
        :param seed_jnt_values: 初期関節角度
        :param max_n_iter: 最大繰り返し回数
        :param toggle_dbg: debug用
        :return:
        """
        iter_jnt_values = seed_jnt_values
        if seed_jnt_values is None:
            iter_jnt_values = self.jlc.get_jnt_values()
        counter = 0
        while True:
            flange_pos, flange_rotmat, j_mat = self.jlc.fk(jnt_values=iter_jnt_values,
              toggle_jacobian=True, update=False)
            f2t_pos_err, f2t_rot_err, f2t_err_vec = rm.diff_between_poses(src_pos=flange_pos,
              src_rotmat=flange_rotmat, tgt_pos=tgt_pos, tgt_rotmat=tgt_rotmat)
            if f2t_pos_err < 1e-4 and f2t_rot_err < 1e-3 and self.jlc.are_jnts_in_ranges(
              iter_jnt_values):
                return iter_jnt_values
            clamped_err_vec = self._clamp_tgt_err(f2t_pos_err, f2t_rot_err, f2t_err_vec)
            delta_jnt_values = np.linalg.pinv(j_mat, rcond=1e-4) @ clamped_err_vec
            if toggle_dbg:
                print("f2t_pos_err ", f2t_pos_err, " f2t_rot_err ", f2t_rot_err)
```

```
                print("clamped_tgt_err ", clamped_err_vec)
                print("coutner/max_n_iter ", counter, max_n_iter)
            if abs(np.sum(delta_jnt_values)) < 1e-6:
                return None
            iter_jnt_values = iter_jnt_values + delta_jnt_values
            if not self.are_jnts_in_range(iter_jnt_values) or counter > max_n_iter:
                return None
            counter += 1
```

この関数の 21 行目で，np.linalg.pinv を呼び出し，NumPy のライブラリ関数を使用して逆行列を計算しています。第 15 行の while 文は，ニュートン-ラフソン法の反復計算ループを実装しています。ループは 18 行目または 29 行目の条件が満たされたときに終了します。 この関数の seed_jnt_values パラメータは，ニュートン-ラフソン法の初期値 $\boldsymbol{q}_0$ です。この値が明示的に指定されていない場合，プログラムは直列リンク機構の現在のコンフィギュレーションを $\boldsymbol{q}_0$ として使用します。 pinv 関数の 12-13 行目は，seed_jnt_values の初期値が明示的に指定されていない場合の処理文です。

dls() に関数は減衰最小二乗法を実装しています。中身は pinv() とはほぼ一致ですが，delta_jnt_values の計算は

```
    delta_jnt_values = np.linalg.inv(j_mat.T @ j_mat + 1e-4 * np.eye(j_mat.shape[1])) @ j_mat.
      T @ clamped_err_vec
```

或いは

```
    delta_jnt_values = np.linalg.lstsq(j_mat, clamped_err_vec, rcond=1e-4)[0]
```

で行います．前の実装は減衰最小二乗法の算式のただのコード化です．後ろの実装は，Numpy の線形方程式系を解く関数 lstsq を利用しています．lstsq 関数はランクの状態を見分けて計算方法を切り替えします．NumPy の pinv 関数の実装は，減衰最小二乗法の実装よりも高速です。また，rcond パラメータを調整することで，特異値が小さすぎる場合の数値的不安定性を回避できます。これにより，WRS システムは修正量を求める際にデフォルトで擬似逆行列法を使用します。

WRS のニュートン-ラフソン法の実装について注意してもらいたいのは，局所線形化の仮定を満たす必要があるため，毎回修正目標 $[\delta\boldsymbol{p}(\delta\boldsymbol{q}_i)\ \delta\boldsymbol{\xi}(\delta\boldsymbol{q}_i]^\top$ の大きさを控える必要があります。 WRS の実装では，この差の大きさを制御するために__clamp_tgt_err メンバー関数を使用しています。修正目標が大きすぎる場合，__clamp_tgt_err 関数は許容される最大値を返します。リスト 4.2-8 には，__clamp_tgt_err 関数の具体的な実装が示されています。

リスト 4.2-8: 修正目標の大きさを控えるための__clamp_tgt_err 関数

```
    def _clamp_tgt_err(self, f2t_pos_err, f2t_rot_err, f2t_err_vec):
        clamped_vec = np.copy(f2t_err_vec)
        if f2t_pos_err >= self.clamp_pos_err:
            clamped_vec[:3] = self.clamp_pos_err * f2t_err_vec[:3] / f2t_pos_err
        if f2t_rot_err >= self.clamp_rot_err:
            clamped_vec[3:6] = self.clamp_rot_err * f2t_err_vec[3:6] / f2t_rot_err
        return clamped_vec
```

KDTree に基づく高速化　ニュートン-ラフソン法の効率と成功率は，初期値 $\boldsymbol{q}_0$ の選択に依存します。効率を向上させるために，WRS システムでは KDTree を用いて初期点を事前に保存する方法を採用しています。この方法の基本的な考え方は以下の通りです。 1. 関節空間をサンプリングし，サンプリングされた関節角度ベクトルを保存する。 2. サンプリングされた関節角度ベクトルに対応する直列リンク機構の先端の位置と回転に対して，KDTree を構築と保存し，最近傍探索を高速化する。 3. ニュートン-ラフソン法を開始する際に，KDTree 内で目標の最近傍位置と回転を検索し，それに対応する関節角度ベクトルを見つけ，その関節角度ベクトルを初期値として使用する。

robot_sim/_kinematics の下にある ik_dd.py は，この KDTree に基づく手法の実装です。 KDTree は事前にデータを保存しているため，この手法をデータ駆動（data driven, dd）と呼ばれ，実装したクラスは DDIKSolver と名付けられた。 DDIKSolver の初期化関数は以下の通りです。

リスト 4.2-9: KDTree に基づく高速化

```
    class DDIKSolver(object):
        def __init__(self, jlc, path=None, identifier_str='test',
```

```
3                       backbone_solver='n', rebuild=False):
4           """
5           :param jlc:
6           :param path:
7           :param backbone_solver: 'n': num ik; 'o': opt ik; 't': trac ik
8           :param rebuild:
9           """
10          self.jlc = jlc
11          current_file_dir = os.path.dirname(__file__)
12          if path is None:
13              path = os.path.join(os.path.dirname(current_file_dir), "_data_files")
14          self._fname_tree = os.path.join(path, f"{identifier_str}_ikdd_tree.pkl")
15          self._fname_jnt = os.path.join(path, f"{identifier_str}_jnt_data.pkl")
16          self._k_bbs = 100  # number of nearest neighbours examined by the backbone solver
17          self._k_max = 200  # maximum nearest neighbours explored by the evolver
18          self._max_n_iter = 7  # max_n_iter of the backbone solver
19          if backbone_solver == 'n':
20              self._backbone_solver = rkn.NumIKSolver(self.jlc)
21          elif backbone_solver == 'o':
22              self._backbone_solver = rko.OptIKSolver(self.jlc)
23          elif backbone_solver == 't':
24              self._backbone_solver = rkt.TracIKSolver(self.jlc)
25          if rebuild:
26              print("Rebuilding the database. It starts a new evolution and is costly.")
27              y_or_n = bu.get_yesno()
28              if y_or_n == 'y':
29                  self.query_tree, self.jnt_data = self._build_data()
30                  self.persist_data()
31                  self.evolve_data(n_times=100000)
32          else:
33              try:
34                  with open(self._fname_tree, 'rb') as f_tree:
35                      self.query_tree = pickle.load(f_tree)
36                  with open(self._fname_jnt, 'rb') as f_jnt:
37                      self.jnt_data = pickle.load(f_jnt)
38              except FileNotFoundError:
39                  self.query_tree, self.jnt_data = self._build_data()
40                  self.persist_data()
41                  self.evolve_data(n_times=100)
```

この関数はデフォルトで {identifier_str}_ikdd_tree.pkl と {identifier_str}_jnt_data.pkl ファイルを開こうとします。これらのファイルはそれぞれ，KDTree とサンプリングされた関節姿勢が保存されていると想定されています。ここでの {identifier_str}は初期化パラメータの値を指します。ファイルが見つかった場合，対応するバックボーンソルバーを呼び出して逆運動学を解きます。バックボーンソルバーには，ニュートン-ラフソン法（NumIKSolver），最適化手法（OptIKSolver），およびハイブリッド手法（TracIKSolver）があります（21-26 行目）。デフォルトのパラメータ設定では，システムは自動的にニュートン-ラフソン法を使用します。ファイルが見つからない場合（40 行目），または rebuild 初期化パラメータが真の値で渡された場合（27 行目），この初期化関数は self._rebuild_data() 関数を呼び出してデータを再構築します。 self._rebuild_data() 関数の具体的な実装は以下の通りです。構築した KDTree とサンプリングされた関節姿勢を戻り値として出力します。

リスト 4.2-10: self._rebuild_data() 関数の実装

```
1   def _build_data(self):
2       # gen sampled qs
3       sampled_jnts = []
4       n_intervals = np.linspace(8, 4, self.jlc.n_dof, endpoint=False)
5       print(f"Buidling Data for DDIK using the following joint granularity: {n_intervals.
          astype(int)}...")
6       for i in range(self.jlc.n_dof):
7           sampled_jnts.append(
8               np.linspace(self.jlc.jnt_ranges[i][0], self.jlc.jnt_ranges[i][1], int(
                  n_intervals[i]+2))[1:-1])
9       grid = np.meshgrid(*sampled_jnts)
10      sampled_qs = np.vstack([x.ravel() for x in grid]).T
11      # gen sampled qs and their correspondent flange poses
12      query_data = []
13      jnt_data = []
14      for id in tqdm(range(len(sampled_qs))):
15          jnt_values = sampled_qs[id]
```

```
            flange_pos, flange_rotmat = self.jlc.fk(jnt_values=jnt_values, toggle_jacobian=
              False)
            # relative to base
            rel_pos, rel_rotmat = rm.rel_pose(self.jlc.pos, self.jlc.rotmat, flange_pos,
              flange_rotmat)
            rel_rotvec = self._rotmat_to_vec(rel_rotmat)
            query_data.append(np.concatenate((rel_pos, rel_rotvec)))
            jnt_data.append(jnt_values)
        query_tree = scipy.spatial.cKDTree(query_data)
        return query_tree, jnt_data
```

WRS システムの JLChain クラスは，デフォルトで DDIKSolver を IK ソルバーとして使用します。そのため，新しい JLChain を定義し，新しい identifier_str が指定された場合，システムは姿勢を再サンプリングし，KDTree を構築と保存します。ユーザーの PyCharm コンソールには，図 12 に示すように進行状況バーが表示され，KDTree と関節姿勢データファイルが生成され，robot_sim/_data_files フォルダに保存されます。この進行状況バーは，新しい identifier_str が指定された場合，または rebuild を強制的に要求された場合にのみ表示されます。それ以外の場合，システムは既存の保存済みファイルを優先して使用します。図 12 に示されているのは，“wrs_cobotta_arm” という名前の Cobotta ロボットクラスのインスタンスが初めて定義された際の結果です。実行が終了した後，robot_sim/_data_files フォルダ内に新たに cobotta_arm_ikdd_tree.pkl および cobotta_arm_jnt_data.pkl の 2 つのファイルが生成されました。同じ名前の Cobotta ロボットクラスのインスタンスを再度定義する際，WRS システムはまずこれらの 2 つのファイルが存在するかどうかを確認します。もし存在すれば，再生成せずに直接再利用します。

```
D:\python311\python.exe E:\wrs\0000_book\list4-2-9-cobotta.py
Known pipe types:
  wglGraphicsPipe
(all display modules loaded.)
  0%|          | 0/40320 [00:00<?, ?it/s]Buidling Data for DDIK using the following joint granularity: [8 7 6 6 5 4]...
100%|██████████| 40320/40320 [00:17<00:00, 2287.11it/s]
new goals::   0%|          | 0/100 [00:00<?, ?it/s]ddik data file saved.
new goals::  41%|████      | 41/100 [00:00<00:01, 43.77it/s]#### Previously unsolved ik solved using the 123th nearest neighbour.
#### Previously unsolved ik solved using the 105th nearest neighbour.
#### Previously unsolved ik solved using the 112th nearest neighbour.
new goals::  79%|███████   | 79/100 [00:02<00:00, 35.99it/s]#### Previously unsolved ik solved using the 122th nearest neighbour.
new goals::  94%|████████  | 94/100 [00:02<00:00, 28.85it/s]#### Previously unsolved ik solved using the 129th nearest neighbour.
```

図 12: DDIKSolver のデータファイルを構築する進行状況バー

図 13 は，前述の KDTree および関節角ベクトルファイルを利用した DDIKSolver による解の結果を示しています。純粋な Python 実装では，この解を求めるのに約 5ms かかり，ある程度リアルタイムの要求を満たすことができます。具体的なコードについては，WRS システムの 0000_book フォルダ内の list4-2-9-cobotta.py ファイルを参照してください。

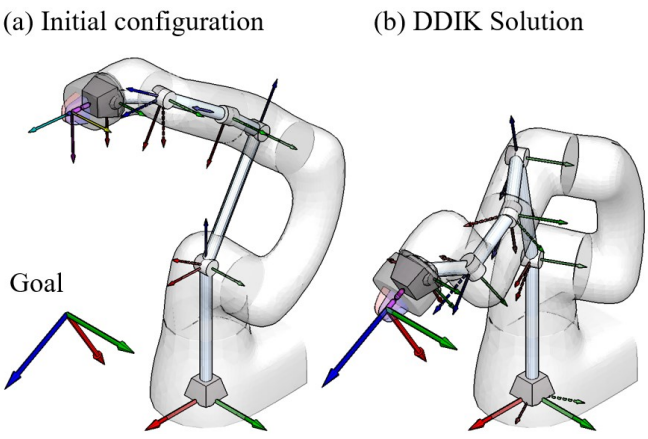

図 13: DDIKSolver による逆運動学の計算

解析法や他の方法の実装　KDTree を使用して初期位置の最適化や処理の高速化を図る方法に加え，ニューラルネットワークを活用して初期位置を予測し，ニュートン-ラフソン法の反復回数を削減することも可能です。WRS システムには現時点ではニューラルネットワークに基づく IK 解法アルゴリズムは含まれていませんが，このような手法を実現したい場合には，ManipulatorInterface インターフェースの ik 関数をオーバーロードして新しいアルゴリズムを実装することができます。

さらに，多くの産業用ロボットでは，設計段階で逆運動学の解法が考慮されており，解析解を直接実現することが可能です。解析解は各ロボットに特化したものであり，WRS システムでは，Universal Robot シリーズ，KHI RS007L，UFactory XArm Lite 6 など，定義済みの一部マニピュレーターに対して解析解の定義が事前に実装されています。これらも本質的には，ManipulatorInterface の ik 関数をオーバーロードすることで実現されています。リスト??には，robot_sim/manipulators/rs007l/rs007l.py で定義された ik 関数のオーバーロード実装が示されています。この関数は，RS007L の手首にある 3 つの関節の回転軸が一点に交わるという特徴を利用し，単純な三角関数関係を通じて逆運動学を直接解いています。このオーバーロードがない場合，システムは DDIK を使用しますが，オーバーロードが実装されている場合は，システムは優先的にオーバーロードされた ik を使用します。

リスト 4.2-11: RS007L の定義にオーバーロードされた IK 関数

```
    def ik(self,
           tgt_pos: np.ndarray,
           tgt_rotmat: np.ndarray,
           **kwargs):
        """
        analytical ik sovler, slover than ddik
        the parameters in kwargs will be ignored
        :param tgt_pos:
        :param tgt_rotmat:
        :return:
        """
        tcp_loc_pos = self.loc_tcp_pos
        tcp_loc_rotmat = self.loc_tcp_rotmat
        flange_rotmat = tgt_rotmat @ tcp_loc_rotmat.T
        flange_pos = tgt_pos - flange_rotmat @ tcp_loc_pos
        rrr_pos = flange_pos - flange_rotmat[:, 2] * np.linalg.norm(self.jlc.jnts[5].loc_pos)
        rrr_x, rrr_y, rrr_z = ((rrr_pos - self.pos) @ self.rotmat).tolist()  # in base coordinate system
        j0_value = math.atan2(rrr_x, rrr_y)
        if not self._is_jnt_in_range(jnt_id=0, jnt_value=j0_value):
            return None
        # assume a, b, c are the axis_length of shoulders and bottom of the big triangle formed by the robot arm
        c = math.sqrt(rrr_x ** 2 + rrr_y ** 2 + (rrr_z - self.jlc.jnts[0].loc_pos[2]) ** 2)
        a = self.jlc.jnts[2].loc_pos[0]
        b = self.jlc.jnts[3].loc_pos[0] + self.jlc.jnts[4].loc_pos[2]
        tmp_acos_target = (a ** 2 + b ** 2 - c ** 2) / (2 * a * b)
        if tmp_acos_target > 1 or tmp_acos_target < -1:
            print("The triangle formed by the robot arm is violated!")
            return None
        j2_value = math.acos(tmp_acos_target) - math.pi
        if not self._is_jnt_in_range(jnt_id=2, jnt_value=j2_value):
            return None
        tmp_acos_target = (a ** 2 + c ** 2 - b ** 2) / (2 * a * c)
        if tmp_acos_target > 1 or tmp_acos_target < -1:
            print("The triangle formed by the robot arm is violated!")
            return None
        j1_value_upper = math.acos(tmp_acos_target)
        # assume d, c, e are the edges of the lower triangle formed with the ground
        d = self.jlc.jnts[0].loc_pos[2]
        e = math.sqrt(rrr_x ** 2 + rrr_y ** 2 + rrr_z ** 2)
        tmp_acos_target = (d ** 2 + c ** 2 - e ** 2) / (2 * d * c)
        if tmp_acos_target > 1 or tmp_acos_target < -1:
            print("The triangle formed with the ground is violated!")
            return None
        j1_value_lower = math.acos(tmp_acos_target)
        j1_value = math.pi - (j1_value_lower + j1_value_upper)
        if not self._is_jnt_in_range(jnt_id=1, jnt_value=j1_value):
            return None
        # RRR
        anchor_gl_rotmatq = self.rotmat
        j0_gl_rotmat0 = anchor_gl_rotmatq @ self.jlc.jnts[0].loc_rotmat
        j0_gl_rotmatq = j0_gl_rotmat0 @ rm.rotmat_from_axangle(self.jlc.jnts[0].loc_motion_ax, j0_value)
        j1_gl_rotmat0 = j0_gl_rotmatq @ self.jlc.jnts[1].loc_rotmat
        j1_gl_rotmatq = j1_gl_rotmat0 @ rm.rotmat_from_axangle(self.jlc.jnts[1].loc_motion_ax, j1_value)
        j2_gl_rotmat0 = j1_gl_rotmatq @ self.jlc.jnts[2].loc_rotmat
        j2_gl_rotmatq = j2_gl_rotmat0 @ rm.rotmat_from_axangle(self.jlc.jnts[2].loc_motion_ax,
```

```
            j2_value)
56          rrr_g_rotmat = (j2_gl_rotmatq @ self.jlc.jnts[3].loc_rotmat @
57                          self.jlc.jnts[4].loc_rotmat @ self.jlc.jnts[5].loc_rotmat)
58          j3_value, j4_value, j5_value = rm.rotmat_to_euler(rrr_g_rotmat.T @ flange_rotmat,
              order='rzxz').tolist()
59          if not (self._is_jnt_in_range(jnt_id=3, jnt_value=j3_value) and
60                  self._is_jnt_in_range(jnt_id=4, jnt_value=j4_value) and
61                  self._is_jnt_in_range(jnt_id=5, jnt_value=j5_value)):
62              return None
63          return np.array([j0_value, j1_value, j2_value, j3_value, j4_value, j5_value])
```

オーバーロードで ik 関数を作り直す場合，クラスの初期化関数から ik_solver パラメータを削除し，数値的な逆運動学の具体的な解法を設定できないようにする必要があります。また，JLChain を定式化する finalize 関数に於いては，ik_solver を None に設定する必要があります。このポイントを理解しやすいため，下記のリスト 4.2-12 と 4.2-13 に RS007L（解析解法）の初期化関数と Cobotta（DDIKSolver を用いた数値解法）の実装を示しています。これらのリストを比較することで，簡単に理解できるでしょう。

リスト 4.2-12: ik 関数をオーバーロードした際の初期化関数

```
1  class RS007L(mi.ManipulatorInterface):
2
3      def __init__(self, pos=np.zeros(3), rotmat=np.eye
         (3), name='khi_rs007l', enable_cc=True):
4          # 省略...
5          self.jlc.finalize(ik_solver=None, identifier_str
             =name)
```

リスト 4.2-13: DDIKSolver を使うときの初期化関数

```
1  class CobottaArm(mi.ManipulatorInterface):
2
3      def __init__(self, pos=np.zeros(3), rotmat=np.eye
         (3), ik_solver='d', name="cobotta_arm", enable_cc=
         False):
4          # 省略...
5          self.jlc.finalize(ik_solver=ik_solver,
             identifier_str=name)
```

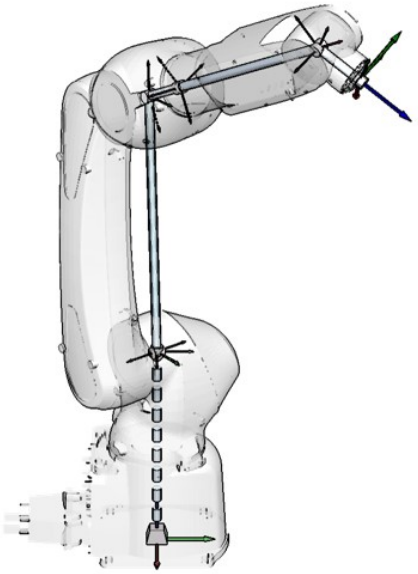

図 14: RS007L の逆運動学の解析解

図 14 は，RS007L の解析的逆運動学による解の結果を示しています。スペースの節約のため，ロボットの初期姿勢は描画していません。解析解の速度は，数値解と比べて遥かに高速です。図 13 の Cobotta における DDIKSolver の計算時間はおおよそ 5ms であり，本図における RS007L の解析的解の計算時間は，Python 11 上で time.perf_counter() を使用して計測した結果，0.3ms でした。

◇ 補足資料：ベクトルの微分 ◇

$\hat{\boldsymbol{x}} = (\boldsymbol{A}^\top \boldsymbol{A})^{-1}\boldsymbol{A}^\top \boldsymbol{b}$ を導出する際，以下の微分を計算しました．

$$\frac{\partial(\boldsymbol{b}^\top \boldsymbol{b} - \boldsymbol{b}^\top \boldsymbol{A}\boldsymbol{x} - \boldsymbol{x}^\top \boldsymbol{A}^\top \boldsymbol{b} + \boldsymbol{x}^\top \boldsymbol{A}^\top \boldsymbol{A}\boldsymbol{x})}{\partial \boldsymbol{x}} = -2\boldsymbol{A}^\top \boldsymbol{b} + 2\boldsymbol{A}^\top \boldsymbol{A}\boldsymbol{x} \tag{71}$$

この計算には以下のベクトル微分公式を利用します．

$$\frac{\partial \boldsymbol{a}^\top \boldsymbol{x}}{\partial \boldsymbol{x}} = \boldsymbol{a} \tag{72}$$

$$\frac{\partial \boldsymbol{x}^\top \boldsymbol{a}}{\partial \boldsymbol{x}} = \boldsymbol{a} \tag{73}$$

$$\frac{\partial \boldsymbol{x}^\top \boldsymbol{A}\boldsymbol{x}}{\partial \boldsymbol{x}} = \boldsymbol{A}^\top \boldsymbol{x} + \boldsymbol{A}\boldsymbol{x} \tag{74}$$

これらの微分公式を式 (71) の左辺に代入して整理すると，右辺の結果が得られます．

$$\begin{aligned} & \frac{\partial(\boldsymbol{b}^\top \boldsymbol{b} - \boldsymbol{b}^\top \boldsymbol{A}\boldsymbol{x} - \boldsymbol{x}^\top \boldsymbol{A}^\top \boldsymbol{b} + \boldsymbol{x}^\top \boldsymbol{A}^\top \boldsymbol{A}\boldsymbol{x})}{\partial \boldsymbol{x}} \\ = \; & \frac{\partial \boldsymbol{b}^\top \boldsymbol{b}}{\partial \boldsymbol{x}} - \frac{\partial \boldsymbol{b}^\top \boldsymbol{A}\boldsymbol{x}}{\partial \boldsymbol{x}} - \frac{\partial \boldsymbol{x}^\top \boldsymbol{A}^\top \boldsymbol{b}}{\partial \boldsymbol{x}} + \frac{\partial \boldsymbol{x}^\top \boldsymbol{A}^\top \boldsymbol{A}\boldsymbol{x}}{\partial \boldsymbol{x}} \\ = \; & -(\boldsymbol{b}^\top \boldsymbol{A})^\top - \boldsymbol{A}^\top \boldsymbol{b} + (\boldsymbol{A}^\top \boldsymbol{A})^\top \boldsymbol{x} + \boldsymbol{A}^\top \boldsymbol{A}\boldsymbol{x} \\ = \; & -2\boldsymbol{A}^\top \boldsymbol{b} + 2\boldsymbol{A}^\top \boldsymbol{A}\boldsymbol{x} \end{aligned} \tag{75}$$

これから，上記の微分公式の導出方法を紹介します．まず，ベクトル微分は以下のように定義されます．

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_1}{\partial x_2} & \cdots & \frac{\partial y_1}{\partial x_n} \\ \frac{\partial y_2}{\partial x_1} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_m}{\partial x_1} & \frac{\partial y_m}{\partial x_2} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} \tag{76}$$

ここで，式 (76) 中のベクトル $\boldsymbol{y}$ と $\boldsymbol{x}$ はそれぞれ m 次元列ベクトルと n 次元列ベクトルと想定します．式 (76) のような記法は分子レイアウト記法と呼ばれます．分子レイアウト記法に対して，以下の分母レイアウト記法もあります．

$$\frac{\partial \boldsymbol{y}}{\partial \boldsymbol{x}} = \begin{bmatrix} \frac{\partial y_1}{\partial x_1} & \frac{\partial y_2}{\partial x_1} & \cdots & \frac{\partial y_m}{\partial x_1} \\ \frac{\partial y_1}{\partial x_2} & \frac{\partial y_2}{\partial x_2} & \cdots & \frac{\partial y_m}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial y_1}{\partial x_n} & \frac{\partial y_2}{\partial x_n} & \cdots & \frac{\partial y_m}{\partial x_n} \end{bmatrix} \tag{77}$$

分子レイアウト記法の行列は，微小変化量間の線形関係を示すため，ヤコビ行列となります。以下のよう

に記号 $\boldsymbol{J}$ で表記されます．ヤコビ行列は一般化座標系での微分計算に使われます．

$$\boldsymbol{J} = \begin{bmatrix} \dfrac{\partial \boldsymbol{f}}{\partial q_1} & \dfrac{\partial \boldsymbol{f}}{\partial q_2} & \cdots & \dfrac{\partial \boldsymbol{f}}{\partial q_n} \end{bmatrix} = \frac{\partial \boldsymbol{f}}{\partial \boldsymbol{q}} = \begin{bmatrix} \dfrac{\partial f_1}{\partial q_1} & \dfrac{\partial f_1}{\partial q_2} & \cdots & \dfrac{\partial f_1}{\partial q_n} \\ \dfrac{\partial f_2}{\partial q_1} & \dfrac{\partial f_2}{\partial q_2} & \cdots & \dfrac{\partial f_2}{\partial q_n} \\ \vdots & \vdots & \ddots & \vdots \\ \dfrac{\partial f_m}{\partial q_1} & \dfrac{\partial f_m}{\partial q_2} & \cdots & \dfrac{\partial f_m}{\partial q_n} \end{bmatrix} \tag{78}$$

一方，分母レイアウト記法の行列は勾配行列と呼ばれます．この資料では使われないため，詳しい説明は割愛します．

ロボットのための微分はヤコビ行列の計算ですので，分子レイアウト記法を用いて演算を行います．ただし，分子あるいは分母の一方がスカラーの場合，記述はベクトルのレイアウトに従います．すなわち，微分する際に，分子がスカラーで分母が縦ベクトルならば，結果は縦ベクトルになります．逆に，分母がスカラーで分子が横ベクトルならば，結果は横ベクトルになります．例えば，以下の式中の $\boldsymbol{x}$ と $\boldsymbol{y}$（太字であることに注意）がそれぞれ縦ベクトルと横ベクトルであると想定すると，レイアウトは右辺のようになります．

$$\frac{\partial y}{\partial \boldsymbol{x}} = \begin{bmatrix} \dfrac{\partial y}{\partial x_1} \\ \dfrac{\partial y}{\partial x_2} \\ \vdots \\ \dfrac{\partial y}{\partial x_n} \end{bmatrix}, \quad \frac{\partial \boldsymbol{y}}{\partial x} = \begin{bmatrix} \dfrac{\partial y_1}{\partial x} & \dfrac{\partial y_2}{\partial x} & \cdots & \dfrac{\partial y_m}{\partial x} \end{bmatrix} \tag{79}$$

公式 (72)∼(74) の分子はすべてスカラーですので，式 (79) を用いて式 (72)∼(74) の左辺を変形すれば，右辺の結果を導出できます．それぞれの導出を順を追って説明します．$\partial \boldsymbol{a}^\top \boldsymbol{x}/\partial \boldsymbol{x}$ の場合，$\boldsymbol{x}$ は縦ベクトルですから，以下のように展開できます．

$$\frac{\partial \boldsymbol{a}^\top \boldsymbol{x}}{\partial \boldsymbol{x}} = \begin{bmatrix} \dfrac{\partial \boldsymbol{a}^\top \boldsymbol{x}}{\partial x_1} & \dfrac{\partial \boldsymbol{a}^\top \boldsymbol{x}}{\partial x_2} & \cdots & \dfrac{\partial \boldsymbol{a}^\top \boldsymbol{x}}{\partial x_n} \end{bmatrix}^\top \tag{80}$$

右辺の任意の要素 $\partial \boldsymbol{a}^\top \boldsymbol{x}/\partial x_i$ に対して，

$$\frac{\partial \boldsymbol{a}^\top \boldsymbol{x}}{\partial x_i} = \frac{\partial \sum_{j=1}^{n} a_j x_j}{\partial x_i} = \frac{\partial a_i x_i}{\partial x_i} = a_i \tag{81}$$

が成り立ちます．式 (81) を用いて式 (80) の各要素を入れ替えると，以下の結果が得られます．

$$\frac{\partial \boldsymbol{a}^\top \boldsymbol{x}}{\partial \boldsymbol{x}} = \begin{bmatrix} \dfrac{\partial \boldsymbol{a}^\top \boldsymbol{x}}{\partial x_1} & \dfrac{\partial \boldsymbol{a}^\top \boldsymbol{x}}{\partial x_2} & \cdots & \dfrac{\partial \boldsymbol{a}^\top \boldsymbol{x}}{\partial x_n} \end{bmatrix}^\top = \begin{bmatrix} a_1 & a_2 & \cdots & a_n \end{bmatrix}^\top = \boldsymbol{a} \tag{82}$$

$\partial \boldsymbol{x}^\top \boldsymbol{a}/\partial \boldsymbol{x}$ の場合，$\boldsymbol{x}^\top \boldsymbol{a} = \boldsymbol{a}^\top \boldsymbol{x}$ ですから，$\partial \boldsymbol{a}^\top \boldsymbol{x}/\partial \boldsymbol{x}$ の場合と同様の計算で式 (73) が得られます．
$\partial \boldsymbol{x}^\top \boldsymbol{A}\boldsymbol{x}/\partial \boldsymbol{x}$ の場合，以下のように展開できます．

$$\frac{\partial \boldsymbol{x}^\top \boldsymbol{A}\boldsymbol{x}}{\partial \boldsymbol{x}} = \begin{bmatrix} \dfrac{\partial \sum_{i=1}^{n}\sum_{j=1}^{n} x_i A_{ij} x_j}{\partial x_1} & \dfrac{\partial \sum_{i=1}^{n}\sum_{j=1}^{n} x_i A_{ij} x_j}{\partial x_2} & \cdots & \dfrac{\partial \sum_{i=1}^{n}\sum_{j=1}^{n} x_i A_{ij} x_j}{\partial x_n} \end{bmatrix}^\top \tag{83}$$

一つの要素に対して演算すると，

$$\frac{\partial \sum_{i=1}^{n}\sum_{j=1}^{n} x_i A_{ij} x_j}{\partial x_k} = \sum_{i=1}^{n} A_{ik} x_i + \sum_{j=1}^{n} A_{kj} x_j \tag{84}$$

が成り立ちます．式 (83) の各要素を入れ替えると，以下の式が得られます．

$$\begin{aligned}
\frac{\partial \boldsymbol{x}^\top \boldsymbol{A}\boldsymbol{x}}{\partial \boldsymbol{x}} &= \left[\sum_{i=1}^{n} A_{i1}x_i + \sum_{j=1}^{n} A_{1j}x_j \quad \sum_{i=1}^{n} A_{i2}x_i + \sum_{j=1}^{n} A_{2j}x_j \quad \cdots \quad \sum_{i=1}^{n} A_{in}x_i + \sum_{j=1}^{n} A_{nj}x_j\right] \\
&= \left[\sum_{i=1}^{n} A_{i1}x_i \quad \sum_{i=1}^{n} A_{i2}x_i \quad \cdots \quad \sum_{i=1}^{n} A_{in}x_i\right] + \left[\sum_{j=1}^{n} A_{1j}x_j \quad \sum_{j=1}^{n} A_{2j}x_j \quad \cdots \quad \sum_{j=1}^{n} A_{nj}x_j\right] \\
&= \boldsymbol{A}^\top \boldsymbol{x} + \boldsymbol{A}\boldsymbol{x}
\end{aligned} \tag{85}$$